Dictionary for

Computer Languages

A.P.I.C. Studies in Data Processing

General Editor: Richard Goodman

Number 1

Some Commercial Autocodes
E. L. Willey, A. d'Agapeyeff, Marion Tribe, B. J. Gibbens, Michelle Clark

Number 2

A Primer of ALGOL 60 Programming
Together with a Report on the Algorithmic Language ALGOL 60
E. W. Dijkstra

Number 3

Input Language for Automatic Progamming Systems
A. P. Yershov, G. I. Khozhukhin, U. M. Voloshin

Number 4

Introduction to System Programming
Edited by Peter Wegner

Number 5

ALGOL 60 Implementation
B. Randall, L. J. Russell

Dictionary for

Computer Languages

A.P.I.C. Studies in Data Processing No. 6

Hans Breuer

Technische Hochschule Darmstadt, Germany

1966

Published for the Automatic Programming Information Centre, Brighton College of Technology, England by Academic Press, London and New York

ACADEMIC PRESS INC. (LONDON) LTD.
24/28 Oval Road
London NW1

U.S. Edition published by
ACADEMIC PRESS INC.
111 Fifth Avenue
New York, New York 10003

Second printing 1973

Library of Congress Catalog Card Number: 66–25680
SBN: 12 132950 X

Reproduced and printed by photolithography and bound in
Great Britain at The Pitman Press, Bath

PREFACE

The increasing complexity of the problems to be solved in science, engineering, business, etc., has led to the development of electronic computers. At the beginning of the sixties, the number of computers started to increase at a very high rate. The last census (*Autom. Data Process. Newsletter* **9**, 14, 1964) shows that more than 40,000 computers are installed or are to be installed very soon on the American continent and in Europe. It also indicates that the yearly rate of increase is 17% for Europe.

With passing time, the computers have grown bigger and more sophisticated and so has the task of programming these machines. Since it is very tedious to use individual machine languages for programming, highly efficient programming languages and appropriate processors have been developed. This development has culminated in the rigid definition of ALGOL 60* in 1963 and of FORTRAN† in 1964. Although many different computer languages have been introduced, clearly the two most important ones are ALGOL 60 and FORTRAN. These languages are also taught at universities, and at engineering and business schools. Extensive libraries of programs written in ALGOL or FORTRAN have been founded, and the exchange of computer programs is growing rapidly. Unfortunately, the overwhelming majority of computers allow only ALGOL or FORTRAN programs to be used, i.e. if a useful FORTRAN program should be obtained and it is thus desirable to run this on a machine with an ALGOL processor, it is necessary to translate it. The aim of the dictionary presented here is not only to assist the programmer during this translation but also to make it possible to translate from ALGOL 60 into FORTRAN (or vice versa) even with a very limited knowledge of computer language itself. To perform this task, the dictionary is arranged in the following way.

Firstly are listed the properties of the different computer types which accept ALGOL and/or FORTRAN programs. (These computers do represent more than 50% of all existing digital machines.‡) Secondly,

* ALGOL 60 is defined in *Num. Math.* **4**, 420 (1963) and also in *Communs Ass. comput. Mach.* **6**, No. 1 (1963). If a reference is made to ALGOL instead of to ALGOL 60, this means that it refers to an ALGOL 60 program or statement which includes input and/or output statements.

† FORTRAN is defined in *Communs Ass. comput. Mach.* **7**, 590 (1964). Throughout the dictionary the traditional notations FORTRAN II (instead of Basic FORTRAN) and FORTRAN IV, abbreviated FIV (instead of FORTRAN) are used. If reference is made to FORTRAN, this means FORTRAN II and FORTRAN IV.

‡ As a rule, the other computers have no computer language at all, or use languages which do not deviate much from either ALGOL 60 or FORTRAN.

the terminology used throughout the dictionary is explained and the input/output statements for different computer processors are discussed. The majority of the book is occupied by the actual dictionary, which is divided into ALGOL 60 → FORTRAN and FORTRAN → ALGOL 60. Short examples of program translation are given in Parts 4 and 5; in Part 6 are given five computer programs and their translations. Four of these programs are complete, including input data and computed results. Part 6 deals also with the problem of how to translate a program in the easiest way.

In Appendices I and II, the reader will find the definitions of ALGOL 60 and of FORTRAN, respectively.

Although it is certainly difficult to present all possible statements, combinations of statements, and their translations in a book of finite length, still the author hopes that he has selected the most useful ones and that this dictionary is of considerable help to everyone who has to work with computer languages.

It is a pleasure for the author to thank the Saskatchewan Branch of IBM for their ready assistance and especially for the permission to use their Selectric typewriter. The help of D. Hutcheon, D. E. Lobb, and H. Theissen in the preparation of the manuscript is gratefully acknowledged. The author is also indebted to the University of Saskatchewan, Saskatoon, and its members for help and hospitality.

Saskatoon
March 1966

H. BREUER

SELECTED LITERATURE ON ALGOL 60, FORTRAN II, AND FORTRAN IV

R. Baumann, "ALGOL-Manual der ALCOR-Gruppe". Oldenbourg, München (1965).

E. W. Dijkstra, "A Primer of ALGOL Programming". Academic Press, London (1962).

D. E. Knuth (ed.), A Proposal for Input-Output Conventions in ALGOL 60. *Communs Ass. comput. Mach.* **7**,273 (1964); Report on Input-Output Procedures for ALGOL 60. *Communs. Ass. comput. Mach.* **7**, 628 (1964).

D. McCracken, "FORTRAN Programming". Wiley, New York (1963); "ALGOL Programming". Wiley, New York (1962).

K. Nickel, "ALGOL-Praktikum". G. Braun, Karlsruhe (1964).

E. I. Organick, "A FORTRAN Primer". Addison-Wesley, Reading, Massachusetts (1963).

B. Randall and L. J. Russell, "ALGOL 60 Implementation". Academic Press, London (1964).

CONTENTS

PREFACE v
SELECTED LITERATURE ON ALGOL 60, FORTRAN II, AND FORTRAN IV vii

Part 1
Computers and their Properties

1.1 List of Computers Considered 3
1.2 List of some Computer Properties 4

Part 2
Definitions of Terms Used in this Dictionary

2.1 List of Abbreviations 11
2.2 List of Definitions 11
2.3 List of Notes 13
2.4 List of Restrictions 17
2.5 List of Computer and Processor Properties which Are Important with Respect to the Translation of Programs 18
2.6 The Different Computer Processors and their Main Restrictions of Advantages with Respect to ALGOL 60 and FORTRAN . . 21
2.6.1 ALCOR group 21
2.6.2 Burroughs Corp. 21
2.6.3 Compagnie BULL 21
2.6.4 Control Data Computer Systems 21
2.6.5 Digital Equipment Corp. 21
2.6.6 Elliott Brothers Ltd. 22
2.6.7 English Electric-Leo 22
2.6.8 Honeywell 22
2.6.9 Olivetti 23
2.6.10 Scientific Data Systems 23
2.6.11 Telefunken 24
2.6.12 Zuse 24
2.7 ALGOL 60 Reference Language Symbols and the Hardware Representation of the Different Computer Types 25

Part 3
Essential Input and Output Statements for Different Computer Processors

3.1 FORTRAN Output Statements and the Appropriate Output Statements of ALGOL Processors 30
3.2 ALGOL Processor Input and Output Statements and the Appropriate FORTRAN Input and Output Statements . . . 36
3.2.1 Knuth's Proposal 37
3.2.2 English Electric-Leo Processor 38
3.2.3 Elliott Brothers Ltd. (NE) Processor 39
3.2.4 ALCOR group 41

Part 1

Computers and their Properties

Part 4

ALGOL 60 → FORTRAN II and IV

ALGOL → FORTRAN 43

Part 5

FORTRAN II and IV → ALGOL 60

FORTRAN → ALGOL 127

Part 6

The Method of Translating a Program

6.1 ALGOL 60 into FORTRAN 187
6.1.1 Translation of the Program into FORTRAN II 187
6.1.2 Translation of the Program into FORTRAN IV 189
6.1.3 The Data 190
6.1.4 A Very Simple Example 191
6.1.5 A Simple Example Including Input Data and Computed Output Values 193
6.1.6 A More Complex Example Including Input Data and Computed Output Values 198
6.2 FORTRAN into ALGOL 60 210
6.2.1 Translation from a FORTRAN II Program. 210
6.2.2 Translation from a FORTRAN IV Program 211
6.2.3 The Data 213
6.2.4 A Simple Example Including Input Data and Computed Output Values 213
6.2.5 A More Complex Example Including Input Data and Computed Output Values 219

Appendix I

Definition of ALGOL 60

Definition of ALGOL 60 223

Appendix II

Definition of FORTRAN

Definition of FORTRAN 257
FORTRAN (FORTRAN IV) 261
BASIC FORTRAN (FORTRAN) 303

1.1 LIST OF COMPUTERS CONSIDERED

Manufacturer	Computer Type
Burroughs Corp., Detroit, Mich., U.S.A.	B 5000, 5500
Compagnie BULL, General Electric, 94 Avenue Gambetta, Paris 20, France	CAB 500, GAMMA 30, 30S, M40, 60
Control Data Computer Systems, 8100 34th Ave. S., Minneapolis 20, Minn., U.S.A.	CDC 160G, 1900, 3100, 3200, 3300, 3400, 3600, 3800, 6400, 6600, 6800, 8090, 8092
Digital Equipment Corp., Maynard, Mass., U.S.A.	PDP-1, 4, 5, 6, 7, 8
Electro-Mechanical Research, Inc. Advanced Scientific Instruments Division, Minneapolis, Min., U.S.A.	ASI 2100, 6020, 6040, 6050, 6060, 6070, 6080
Elliott Brothers Ltd., Borehamwood, Hertfordshire, England	NE 503, 803, 803B
English Electric-Leo/Computers Ltd., Kidsgrove, Staffordshire, England	KDF 9
Ferranti Electronics, Toronto, Canada	FP 6000
General Electric Company Computer Department, P.O. Box 270, Phoenix, Arizona, U.S.A.	GE 205, 235, 415, 425, 435, 600
Honeywell Electronic Data Processing, 60 Walnut Street, Wellesley Hills 81, Mass., U.S.A.	H 21, 22, 200, 400, 800, 1400, 1800
International Business Machines Company Inc., 112 East Post Road, White Plains, New York, U.S.A.	IBM 360, 704, 709, 1401, 1410, 1620, 1800, 7040, 7044, 7070, 7090, 7094
International Computers and Tabulators Ltd., London, S.W.15, England	ICT 1902, 1903, 1904, 1905, 1906, 1907, 1909
Olivetti, Milano, Via Pirelli 32, Italy	Elea 4001, 6001

Manufacturer	Computer type
Scientific Data Systems, Santa Monica, Calif., U.S.A.	SDS 910, 920, 925, 930, 9300
Telefunken AG, Konstanz, Germany	TR 4, 10
Zuse KG, Bad Hersfeld, Germany	Z 22, 23, 25

1.2 LIST OF SOME COMPUTER PROPERTIES

Computer	Core storage capacity [words]	Word length [bits]	Cycle time [μsec]	ALGOL proc.	FORTRAN II proc.	FORTRAN IV proc.
ASI 2100	4K–8K	21	2	No	Yes	No
ASI 6020 6040 6050	4K–32K	24	1.9	No	Yes	No
ASI 6060 6070 6080	4K–32K	24	1.9	No	Yes	No
B 5000	4K–32K	48	6	Yes	No	No
B 5500	32K	48	0.125	Yes	Yes	Yes
CAB 500	16K	32	160	Yes	Yes	No
CDC 160G	8K–128K	13	1.35	No	Yes	No
CDS 1900	4K–32K	16	1.2	No	No	Yes
CDC 3100	4K–32K	24	1.75	No	Yes	Yes
CDC 3200	8K–32K	24	1.25	No	Yes	Yes
CDC 3300	8K–128K	24	0.8	No	Yes	Yes
CDC 3400	16K–32K	48	1.5	Yes	No	Yes
CDC 3600	16K–256K	48	1.4	Yes	No	Yes
CDC 3800	16K–256K	48	0.8	Yes	No	Yes

K = 1024

Computer	Core storage capacity [words]	Word length [bits]	Cycle time [μsec]	ALGOL proc.	FORTRAN II proc.	FORTRAN IV proc.
CDC 6400	32K–128K	60	1.0	Yes	No	Yes
CDC 6600	64K–128K	60	0.5	Yes	No	Yes
CDC 6800	32K–128K	60	0.1	Yes	No	Yes
CDC 8090	4K–32K	12	6.4	No	Yes	No
CDC 8092	2K–4K	8	4.0	No	Yes	No
Elea 4001				Yes¶	Yes	No
Elea 6001				No	Yes	No
FP 6000	4K–32K	24	2	Yes	No	No
GAMMA 30 30S	40K	6	4.8	Yes	Yes	No
GAMMA M40	32K	24	5	Yes	No	Yes
GAMMA 60	32K	24	10	Yes	No	No
GE 205	4K–16K	20	36.0	No	Yes	Yes
GE 235	4K–16K	20(40)	6.0	No	Yes	Yes
GE 415	4K–128K	24	9.2	No	Yes	No
GE 425	4K–128K	24	5.1	No	Yes	No
GE 435	4K–128K	24	2.7	No	Yes	No
GE 600	32K–256K	36	2.0	No	Yes	Yes
H 21	2K–16K	18	6	No	No	No
H 22	2K–16K	18	1.75	No	No	No
H 200	2K–32K*		2†	No	No	Yes
H 400	1K–4K	48	18.5	No	Yes	No
H 800	8K	48	6	No	Yes	No
H 1400	4K	48	6.5	No	Yes	No

K = 1024 * Characters † Per character ¶ A very limited processor.

Computer	Core storage capacity [words]	Word length [bits]	Cycle time [μsec]	ALGOL proc.	FORTRAN II proc.	FORTRAN IV proc.
H 1800	16K	48	2	No	No	Yes
IBM 360–30	2K–16K	32	6	No	No	Yes
IBM 360–40	4K–64K	32	5	No	No	Yes
IBM 360–50	16K–65K‡	32	2	No	No	Yes
IBM 360–60	32K–128K‡	32	1	No	No	Yes
IBM 360–62 360–70	64K–128K‡	32	0.5	No	No	Yes
IBM 704 709	4K–32K	36	12	No	Yes	No
IBM 1401	1.4K–16K*	Var.	11.5†	No	Yes	No
IBM 1410	10K–80K*	Var.	4.5†	No	No	Yes
IBM 1620_{I}	20K–80K*	Var.	20†	No	Yes	No
IBM 1620_{II}	20K–80K*	Var.	10†	No	Yes	No
IBM 1800	4K–32K	16	2	No	No	Yes
IBM 7040	4K–32K	36	8	No	No	Yes
IBM 7044	4K–32K	36	2.5	No	No	Yes
IBM 7070	5K–30K	40	6	No	Yes	No
IBM 7074	5K–30K	40	4	No	Yes	No
IBM 7090	32K	36	2.18	No	Yes	Yes
IBM 7094	32K–64K	36	1.4	No	Yes	Yes
ICT 1902	4K–16K	48	6.0	Yes	Yes	No
ICT 1903	8K–32K	48	2.0	Yes	Yes	No
ICT 1904	8K–32K	48	2.0	Yes	Yes	No
ICT 1905	8K–32K	48	2.0	Yes	Yes	No

K = 1024 * Characters † Per character ‡ Up to 64000K possible.

Computer	Core storage capacity [words]	Word length [bits]	Cycle time [μsec]	ALGOL proc.	FORTRAN II proc.	FORTRAN IV proc.
ICT 1906	32K–256K	48	1.1	Yes	Yes	No
ICT 1907	32K–256K	48	1.1	Yes	Yes	No
ICT 1909	16K–32K	48	6.0	Yes	Yes	No
KDF 9	8K–32K	48	6	Yes	Yes	No
NE 503	8K–128K	38	3.5	Yes	No	No
NE 803 803B	4K–8K	38	6	Yes	No	No
PDP-1	4K–32K	18	5	Yes	Yes	No
PDP-4	4K–32K	18	8	No	Yes	No
PDP-5	4K–32K	12	6	No	Yes	No
PDP-6	8K–256K	36	5.2 or 0.5	No	Yes	Yes
PDP-7	4K–32K	18	1.75	No	Yes	No
PDP-8	4K–32K	12	1.6	No	Yes	No
SDS 910	2K–16K	24	8	No§	Yes	No
SDS 920	4K–16K	24	8	No§	Yes	No
SDS 925	4K–16K	24	1.75	No§	Yes	No
SDS 930	4K–32K	24	2	No§	Yes	No
SDS 9300	4K–32K	24	1.75	No§	Yes	Yes
TR 4	12K–28K	48	6	Yes	No	Yes
TR 10	10K–80K	6	8	Yes	No	No
Z 22				Yes	No	No
Z 23	4K–8K	38	18	Yes	No	No
Z 25	5K–20K	18		Yes	No	No

K=1024 § May be available in summer 1966.

Part 2

Definition of Terms Used in this Dictionary

2.1 LIST OF ABBREVIATIONS

α Maximum number of characters which can be used to identify a variable identifier or a subprogram identifier.

β Maximum value for a subscript.

γ Number of continuation cards permitted.

δ Maximum number of significant digits for an integer number (or range of integer numbers).

ϵ Maximum number of significant digits for a real number.

ζ Maximum number for a label (statement number).

η Range of permissible numbers in exponents of real numbers.

w The width of field in a format statement. (This includes the numbers, the signs, the decimal point, and the blank spaces to provide spacing between numbers.)

d Number of decimal positions which appear to the right of the decimal point.

b Number of blank fields (spaces) appearing before a value.

e, c Positive integer numbers, used in connection with input/output statements.

2.2 LIST OF DEFINITIONS

$a, a_1, a_2, \ldots, a_k$	Arbitrary identifiers for (subscripted or non-subscripted) variables and/or constants and/or values. They must consist of at least one but not more than α characters, the first must be an alphabetic character. Additional restriction in FORTRAN II: the last character of an identifier must not be the letter F.
$Na, Na_1, Na_2, \ldots, Na_k$	The preceding N emphasizes that the variable or constant must be of integer type.
$Aa, Aa_1, Aa_2, \ldots, Aa_k$	The preceding A emphasizes that the variable or constant must be of real type.
$c, c_1, c_2, \ldots, c_k$	Arbitrary constants.
$Nc, Nc_1, Nc_2, \ldots, Nc_k$	The preceding N emphasizes that the constant must be of integer type.
$Ac, Ac_1, Ac_2, \ldots, Ac_k$	The preceding A emphasizes that the constant must be of real type.

$v, v_1, v_2, \ldots, v_k$	Arbitrary variable identifiers.
$Nv, Nv_1, Nv_2, \ldots, Nv_k$	The preceding N emphasizes that the variable must be of integer type.
$Av, Av_1, Av_2, \ldots, Av_k$	The preceding A emphasizes that the variable must be of real type.
m_1, m_2, m_3, m_4	Nonsubscripted integer variables or constants (if not otherwise indicated they are positive).
$n, n_1, n_2, \ldots, n_k$	Statement labels (numbers). Any number from 1 through ζ can be used. In FORTRAN the statement number must appear in column (1) 2 through 5, this is indicated by using a vertical line after the number. (Examples: *21*| or *9945*| or *5*| or *99929*|.)
token, token1, token2	Arbitrary identifiers for a subprogram. The identifier must begin with an alphabetic character. It must consist of at least one, but not more than α characters. In FORTRAN II the first character of the identifier determines the type of the subprogram. In FORTRAN IV the first character can determine the type of the subprogram.
Ntoken, Ntoken1, Ntoken2	The preceding N indicates that the result of the subprogram is of integer type.
Atoken, Atoken1, Atoken2	The preceding A indicates that the result of the subprogram is of real type.
term, term1, term2	Arbitrary expressions.
Nterm, Nterm1, Nterm2	The preceding N indicates that the expression is of integer type.
Aterm, Aterm1, Aterm2	The preceding A indicates that the expression is of real type.
Lterm	The preceding L indicates that the expression is of logical (Boolean) type.
text, text1, text2	Any sequence of characters to explain a program or a part of it or to give a comment. This could be one character.
matrix, matrix1, matrix2	Arbitrary identifiers for an array. Obeys the same rules as a variable identifier.

Nmatrix, Nmatrix1, Nmatrix2	The preceding *N* emphasizes that the array identifier must be of integer type.
Amatrix, Amatrix1, Amatrix2	The preceding *A* emphasizes that the array identifier must be of real type.
jump	Arbitrary identifier for a switch. It obeys the same rules as a variable identifier.
{ }, {.}, {..}	Symbolizes one of the FORTRAN format settings *Ew.d*, *Fw.d*, *Iw*, *cH* or *Ac* (or their translation).
⋮⋮⋮ ⋮⋮ ⋮	Symbolizes arbitrary parts of a program.
... ... *term* ... or ... *term* ... ;	The periods indicate that *term* is a part of a statement. The periods do not indicate whether *term* actually is placed at the beginning, or at the end, or in the middle of the statement.
┌──────────┐	This indicates input and/or output values in examples.

2.3 LIST OF NOTES

N1 The type (real or integer) of the result is determined by the type of the argument. The type of the argument must be as indicated. If this is not the case a preceding statement must be used to change the type in the proper way.

N2 The variables and/or constants must all be of the same type, the type is indicated, i.e. real or integer. If this is not the case a preceding statement must be used to change the type in the proper way.

N3 Instead of *N*, the variable identifiers may begin also with *I*, *J*, *K*, *L*, or *M*.

N4 Instead of *A*, the variable identifiers may also begin with another alphabetic character, except *I*, *J*, *K*, *L*, *M*, or *N*.

N5 The last 1 (in 1.0000001) should appear ($\epsilon - 1$) places to the right of the decimal point.

N6 In this case m_1, m_2, and m_3 include the (negative or positive) sign.

N7 **end** or **:** of a for statement in an ALGOL 60 program is always translated with *n*| *CONTINUE*.

N8 For details on the format statement see under *FORMAT*.

N9 The following combinations cannot be used as identifiers for variables, switches, labels, or procedures:

abs, arctan, array, cos, begin, comment, cos, do, else, end, entire, equal, exit, exp, for, go to, greater, if, integer, label, less, ln, not equal, not greater, not less, power, print, procedure, read, real, sign, sin, sqrt, step, then, until, value.

N10 The following combinations cannot be used as identifiers for variables, functions or subroutines:
ABS, ACCEPT, ASSIGN, ATAN, CALL, COMMON, COS, DATA, DO, END, ENDFILE, EXIT, EXP, FETCH, FIND, FORMAT, GO TO, IF, INTEGER, LOG, OUTPUT, PAUSE, PRINT, PUNCH, READ, REAL, RETURN, REWIND, SIN, SQRT, STOP, TAPE, TYPE, WRITE, WRITEDR, XABS, XEXP, XLOG, XSQRT, and identifiers with *F* as the last letter. (The last rule is not valid for arithmetic statement functions.)

N11 An arbitrary expression can replace a_1 and/or a_2.

N12 These characters can be replaced by *term*.

N13 Each of these characters can be negative or positive or zero.

N14 These variable identifiers must be declared in the head of the program under **integer** if they are of integer type or under **real** if they are of real type.

N15 If there is no special statement specifying in which form the values should appear in the output data, the values appear as decimal numbers with ϵ significant digits times a power of ten. Integers appear without decimal point.

N16 The same as above, only replace *FUNCTION Ntoken* by *INTEGER FUNCTION Ntoken*.

N17 When an **if** statement follows directly an **if . . . then** statement the following **if** statement must be placed between **begin** and **end**.

N18 When an if statement follows directly a for statement, then the if statement must be placed between **begin** and **end**.

N19 This is a dummy statement if *jump* [. . .] is not defined.

N20 This is a dummy statement if the label is not defined.

N21 The *COMMON* statement must contain all the variable identifiers and/or array identifiers $(v_i, \ldots, v_j)$ which are not mentioned as arguments of the subprogram and which are not defined inside the subprogram. A *COMMON* statement with arguments of the same number, order, and type must also appear in the main program preceding the first appearance of those arguments. An

array identifier appearing in a *COMMON* statement must be preceded by the appropriate *DIMENSION* statement.
If during the translation more than one *COMMON* statement becomes necessary it is recommended that all these different statements be placed in one *COMMON* statement.

N22 Instead of a comma the arguments can be separated by)*text*:(This does not affect the translation.

N23 The part of the subprogram ⋮⋮⋮ must be treated like a separate program, i.e. it has to begin with a declaration (type, array, switch, procedure) for all local identifiers. Local identifiers are all identifiers which do not appear in the *COMMON* statement or as arguments of the subprogram.

N24 Those arguments of the subprogram which are variable identifiers should follow the indicator **value**. (This is not always necessary, but it simplifies the task of the processor.)

N25 The real and/or integer arguments of the subprogram which are variable identifiers should follow the indicators **real** and/or **integer** respectively. (This is not always necessary, but it simplifies the task of the processor.)

N26 If the array identifier appears also as an argument of a subprogram, then see, for example:
SUBROUTINE token (. . , . . , *Nmatrix*, . . , . .)

N27 There may, or may not be a *COMMON* statement.

N28 The same as above, only replace *FUNCTION Atoken* by *REAL FUNCTION Atoken.*

N29 The part of the subprogram ⋮⋮⋮ must be treated like a separate program.

N30 This type declaration overrides only the normal mode indication, it does not necessarily include all (integer and/or real) variable identifiers.
In contrast to FORTRAN IV, the type declarations in an ALGOL 60 program must contain all (integer and/or real) variable identifiers.

N31 The main program must contain the declaration *INTEGER token.* This declaration must precede the first call for the *INTEGER FUNCTION.*

N32 The main program must contain the declaration *REAL token.* This declaration must precede the first call for the *REAL FUNCTION.*

N33 *Jump* appeared in a preceding **switch** *jump*: $= v_1, v_2, \ldots, v_k$ declaration.

N34 The statement on the right side of the logical if statement cannot be a do statement or a logical if statement.

N35 The main program must contain the declaration *LOGICAL token.* This declaration must precede the first call for the *LOGICAL FUNCTION.*

N36 The real and/or integer and/or logical arguments of the subprogram which are variable identifiers should follow the indicators **real** and/or **integer** and/or **Boolean** respectively. (This is not always necessary, but it simplifies the task of the processor.)

N37 If . . . and/or ⋮⋮⋮ contain more than one statement, then translate according to:

```
      IF(....) GO TO n1
      IF(....) GO TO n2
      GO TO n
n1|   ...
      GO TO n
n2|   ⋮⋮⋮
 n|   CONTINUE
```

N38 If ⋮⋮⋮ and/or ⁝⁝⁝ contain more than one statement, then translate according to:

```
      IF(....) GO TO n1
      GO TO n2
n1|   ⋮⋮⋮
      GO TO n
n2|   ⁝⁝⁝
 n|   CONTINUE
```

N39 If . . . contains more than one statement, then translate according to:

```
      IF(....) GO TO n1
      GO TO n
n1|   ...
 n|   CONTINUE
```

N40 The range of a **for . . . do** statement goes up to the next semicolon or includes the constants of the immediately following **begin . . . end** brackets.

N41 The function is calculated with a precision equivalent to twice as many significant digits as are obtained in ordinary operation.

N42 Some FORTRAN II processors such as IBM 1620_{II} require that the letter *F* be the last letter of the identifier of a standard function.

N43 The argument of *ABS*(. . .) must be of real type. If the argument is of integer type, then replace the right-hand side with:

$$IABS(\ldots) + 1$$

Exception: IBM 1620_{II}:
In this case replace the right-hand side with:

$$ABSF(\ldots) + 1$$

N44 Add the contents of the bracket and use this value.

2.4 LIST OF RESTRICTIONS

R1 The argument must be of the type real. The result is of the same type.

R2 The variable identifier cannot begin with *I*, *J*, *K*, *L*, *M*, or *N*.

R3 The length of the statement or text is limited to γ continuation cards.

R4 A *FORMAT* statement cannot be the first statement in a *DO* loop.

R5 The identifier of a FORTRAN II subprogram cannot end with the character *F*.

R6 Not all processors for IBM 1620_{I} and IBM 1401 allow *SUBROUTINE* and *FUNCTION* statements.

R7 In FORTRAN it is not possible to have a variable identifier (or *term*) act as a bound in an array declaration. If this occurs in an ALGOL program, then one should replace the variable identifier (or *term*) by an estimated integer number.

2.5 LIST OF COMPUTER AND PROCESSOR PROPERTIES WHICH ARE IMPORTANT WITH RESPECT TO THE TRANSLATION OF PROGRAMS

Computer	α	β	γ	δ	ϵ	ζ	η
2100 ASI 6020 6040	6	32767	9	−8388608 +8388608	11	99999	−76 +76
B 5000	10			11	8		−46 +69
B 5500	30	1023	0	16	16		−63 +63
CAB 500	10,5*	32767		7	8	85	
8090 CDC 160G 8092	6		9	6	8	99999	−32 +32
3100 CDC 3200 3300	8		4	6	10	32767	−308 +308
3400 CDC 3600 3800	8		‡	14	10	99999	−308 +308
6400 CDC 6600 6800	8	131071	§	17	14	99999	−308 +308
CDC 1900					9		
Elea 4001 6001	5	99999	†	5	8	9999	−49 +50
FP 6000							
GAMMA 30 30S	6			8	8		−99 +99
GAMMA M40	6	32767	9	7	12	9999	−77 +77

* ALGOL proc.
† Tape.
‡ A statement may have up to 598 operators, deliminators, and identifiers.
§ A statement may have up to 660 operators, deliminators, and identifiers.

Computer	α	β	γ	δ	ε	ζ	η
GAMMA 60	8			10	10		−40 +40
H 21							
H 22							
H 200	6	28672	9	2–20	2–20	99999	−99 +99
H 400	6	3072	3	13	10	99999	−64 +63
H 800	6	32767	9	13	12	32767	−76 +76
H 1400	6	6144	6	13	9	99999	−50 +49
H 1800	6	28672	19	13	18	32767	−76 +76
IBM 360	6	9999	9	17	15	99999	−78 +75
IBM 704 709	6	32767	9	−131071 +131071	8	32768	−38 +38
IBM 1401 1410	6	9999	9	2–20 2–40	2–20 2–40	99999	−50 +49
IBM 1620_{I}	5	9999	0	4	8	9999	−50 +49
IBM 1620_{II}	6	9999	4	4–20	2–28	9999	00 100
IBM 1800	5	9999	5	−32767 +32767	10	99	−99 +99
IBM 7040 7044	6	9999	9	11	9	99999	−38 +38
IBM 7070 7074	6	9999	9	10	8	99999	−50 +49
IBM 7090 7094	6	32767	19	−131071 +131071	8	32768	−38 +38

Computer	α	β	γ	δ	ε	ζ	η
KDF 9	8		†	13	13	1000	−128 +128
503 NE 803 803B	6		†	11	8		−76 +76
PDP-1	6	256144	†	10	10	256144	−99 +99
PDP-4	6	256144	†	−131071 +131071	10	99999	−99 +99
PDP-5	4	4096	†	−2047 +2047	7	2047	−2047 +2047
PDP-6	6	256144	9	10	16	99999	−131071 +131071
PDP-7	6	256144	†	−131071 +131071	10	99999	−99 +99
PDP-8	4	4096	†	−2047 +2047	7	2047	−2047 +2047
910 920 SDS 925 930 9300	8	32767	9	−8388607 +8388607	12	99999	−77 +77
TR 4 10	6	30000		13	20	99999	−150 +150
22 Z 23 25	6	255	†	11	9		−38 +39

† Tape.

2.6 THE DIFFERENT COMPUTER PROCESSORS AND THEIR MAIN RESTRICTIONS OR ADVANTAGES WITH RESPECT TO ALGOL 60 AND FORTRAN

2.6.1. ALCOR-Group

1. The separator **while** is not used.
2. The declarator **own** is not used.
3. A variable identifier may have an arbitrary number of characters, but only the first 6 are used to identify it.
4. The arithmetic operator $\div$ is not used.
5. If the exponent in an exponential operation is a positive integer, then the result is of the same type as the base.
6. Only simple Boolean expressions are used.
7. The controlled variable in a **for** statement cannot be a subscripted variable.
8. All formal parameters of a subprogram must be specified.
9. **go to** an undefined **switch** is not allowed.

2.6.2 Burroughs Corp

The ALGOL 60 language is fully implemented.

2.6.3 Compagnie BULL

ALGOL processor:

1. The declarator **own** is not used.
2. The specifier **string** is not used.
3. All formal parameters of a subprogram must be specified.
4. Labels must not be unsigned integers.

2.6.4 Control Data Computer Systems

FORTRAN II processor:

Besides the normal features, this processor includes magnetic tape control statements.

FORTRAN IV processor:

1. More than one statement may be written on one record by using the statement separator $.
2. The last character of a standard library function is an F.

2.6.5 Digital Equipment Corporation

FORTRAN II processor:

The FORTRAN II processor for the machine types PDP-5 and PDP-7 includes all FORTRAN IV features with the exception of the logical expressions.

2.6.6 Elliott Brothers

1. A variable identifier may have an arbitrary number of characters, but only the first 6 are used to identify it.
2. The result of exponentiation is always real.
3. Labels must not be unsigned integers.
4. A **go to** statement inside a procedure body cannot lead to a statement outside this body.
5. Any **label** used to label a statement must be declared at the head of the innermost block and be included in the switch list of a **switch** declaration.
6. All formal parameters of a subprogram must be specified.
7. A switch identifier or a subprogram identifier may not appear as an actual parameter of a subprogram.
8. The logical operator $a_1 \supset a_2$ must be replaced by a_2 **or not** a_1.

2.6.7 English Electric-Leo

1. All formal parameters of a subprogram must be specified.
2. Labels must not be unsigned integers.
3. **go to** an undefined **switch** produces an error indication.
4. A **real procedure** or **integer procedure** must contain an assignment to the **real procedure** or **integer procedure** identifier.
5. The declaration **own array** is not allowed.

2.6.8 Honeywell

FORTRAN II processor (called Automath 400):

In addition to the normal features, this processor includes:

1. FORTRAN IV statements in connection with the use of magnetic tape as input and output devices.
2. An ERASE statement clears to zero the locations corresponding to the identifiers specified in a list.

FORTRAN IV processor (called Automath 800):

In addition to the normal features, this processor includes:

1. It is possible to interspace H800 instructions, written in the ARGUS language, into the FORTRAN IV program.
2. A BUFFER statement makes it possible to overlap reading and/or writing and/or computation.
3. An ERASE statement clears to zero the locations corresponding to the identifiers specified in a list.

2.6.9 Olivetti

ALGOL processor (called PALGO):

1. Expressions used as subscripts can be only linear combinations of integer variable identifiers and integers.
2. The main program and the subprograms consist only of one block.
3. Compound statements are accepted only if controlled by **for** statements.
4. Compound conditional statements are not allowed.
5. Non-local variables and parameters called by identifiers cannot appear on the left side of an assignment statement.

FORTRAN II processor:

Besides the normal features this processor includes magnetic tape control statements.

2.6.10 Scientific Data Systems

ALGOL processor: No information available (At present).

FORTRAN II processor:

In addition to the normal features, this processor includes:

1. Expressions of mixed type are allowed.
2. The index, initial value, limit value, and increment value of a DO statement can be of real type.
3. The increment value of a DO statement can be negative and can be changed inside the loop.
4. FORTRAN IV statements in connection with the use of magnetic tape as input and output devices.
5. Subscripts can be negative or zero.
6. An array may have more than three dimensions. Upper and lower bounds can be introduced, they may be positive, zero, or negative.

Example: DIMENSION matrix(−10/10,0/5,12,7/45).

FORTRAN IV processor:

In addition to the normal features, this processor includes:

1. Expressions of mixed type are allowed.
2. The index, initial value, limit value, and increment value of a DO statement can be of real type.
3. The increment value of a DO statement can be negative and can be changed inside the loop.
4. A subscript can be negative, zero, or a general expression.
5. An array may have more than three dimensions. Upper and lower bounds can be introduced, they may be positive, zero, or negative.

Example: DIMENSION matrix(−10: 10,0: 5,12,7: 45).

6. Dynamic arrays:

Example: ALLOCATE matrix(K: I/J,SQRT(P)).

7. Subprograms may be compiled with each other and with the main program. In this case a COMMON statement is not necessary.

8. Local statement labels, i.e. these labels are restricted to a given program area. Program areas are limited by using the statement END LABELS.

2.6.11 Telefunken

ALGOL processor:

The same restrictions as for the ALCOR group.

2.6.12 Zuse

The same restrictions as for the ALCOR group.

2.7 ALGOL 60 REFERENCE LANGUAGE SYMBOLS AND THE HARDWARE REPRESENTATION OF THE DIFFERENT COMPUTER TYPES

	B 5000	ALCOR	NE	KDF9	BULL CAB 500	BULL GAMMA	CDC
Arithmetic operators							
+	+	+	+	+	+	+	+
−	−	−	−	−	−	−	−
×	×	×	*	×	×	*	*
/	/	/	/	/	/	/	/
÷	DIV	n.u.	DIV	*DIV	°	%	//
↑	*	POWER	**	**	*	Δ	**
Relational operators							
<	<	LESS	LESS	*≥	<	<	'LS'
≤	≤	NOT GREATER	LESSEQ	*>	≤	<=	'LQ'
=	=	EQUAL	EQUAL	=	=	=	'EQ'
≥	≥	NOTLESS	GREQ	≥	≥	>=	'GQ'
>	>	GREATER	GR	>	>	>	'GR'
≠	≠	NOT EQUAL	NOTEQ	≠	≠	≠	'NQ'
Logical operators							
≡	EQV	EQUIV	EQUIV	*EQV	EQUI	EQUI	'EQV'
⊃	IMP	IMPL	n.u.	*IMP	IMPLI	IMPL	'IMP'
V	OR	OR	OR	*OR	OR	OU	'OR'
Λ	AND	AND	AND	*AND	AND	ET	'AND'
⁄	NOT	NOT	NOT	*NOT	NOT	NON	'NOT'
Separators							
,	,	,	,	,	,	,	,
.	.	.	.	.	.	.	.
10	10*	10	@	ν	¶	¶ or 10	′
:	:	:	:	→	:	:	. .
:=	←	:=	:=	*=	:=	:=	=
;	;	;	; or ′	*,	Δ	<> or ;	$
Brackets							
( )	( )	( )	( )	( )	( )	( )	( )
[ ]	[ ]	[ ]	( )	*(*)	·‖·	$( )$	(/ /)
‘ ’		′ ′	$?	*Q *U	n.u.	″ ″	″ ″
begin	BEGIN	BEGIN	BEGIN	*BEGIN	DEBUT	DEBUT	'BEGIN'
end	END	END	END	*END	FIN	FIN	'END'
Alphabetic characters							
a–z A–Z	A–Z	A–Z	A–Z	A–Z	A–Z	A–Z	A–Z
Numeric characters							
0–9	0–9	0–9	0–9	0–9	0–9	0–9	0–9

n.u. not used.

Part 3

Essential Input and Output Statements for Different Computer Processors

One of the major differences between the two languages FORTRAN and ALGOL 60 is that FORTRAN includes input/output statements and ALGOL 60 does not. Thus, for every ALGOL processor a decision is necessary as to how the computer may communicate with the outside world. Some computer manufacturers have incorporated the FORTRAN input/output statements into their ALGOL processors, while others have developed their own input/output systems (which in most cases are easier to handle than the FORTRAN input/output statements).

In the middle of 1964 the ALGOL committee published a proposal for input/output conventions in ALGOL 60* (known as the Knuth proposal) which in the near future will very likely be incorporated into the existing ALGOL processors.

The following table shows the different computer manufacturers and the input/output system used:

Advanced Scientific Instruments	FORTRAN
Burroughs Corp.	Own system
Compagnie BULL	Own system†
Honeywell	FORTRAN
International Business Machines	FORTRAN
English Electric-Leo	Own system, very close to Knuth's proposal†
Elliott Brothers Ltd.	Own system
Digital Equipment Corp.	FORTRAN
Scientific Data Systems	FORTRAN
Telefunken AG	ALCOR group, to be changed into Knuth's proposal†
Zuse	ALCOR group
Olivetti	FORTRAN
Control Data Computer Systems	Own system†
International Computers and Tabulators Ltd.	Own system, to be changed into Knuth's proposal.†
General Electric Company	FORTRAN

Since it is impossible to present all the different input/output systems in full detail, we restrict ourselves to the following points:

1. The FORTRAN input/output statements are explained in Section 5.0. Appropriate examples and the translation (if possible) into input/output statements of the NE (NE 803) are given.

2. In Section 3.1 essential FORTRAN output statements are translated as exactly as possible into the corresponding statements of different ALGOL processors (including Knuth's proposal).

3. In Section 3.2 essential input/output statements of different ALGOL processors (including Knuth's proposal) are translated as exactly as possible into the corresponding FORTRAN statements.

* *Communs Ass. comput. Mach.* **7**,273(1964) and **7**,628(1964).

† Of course, the FORTRAN processor uses the FORTRAN input/output system.

3.1 FORTRAN OUTPUT STATEMENTS AND THE APPROPRIATE OUTPUT STATEMENTS OF ALGOL PROCESSORS

On the following pages some essential FORTRAN output statements are translated as closely as possible into the appropriate statements of Knuth's proposal, the KDF9 processor, and the NE processor.

We restrict ourselves to output statements, because it is very simple to read in a value with the standard input of an ALGOL processor.

For simplification the paper tape punch was chosen as output device for all statements mentioned. Where necessary, examples are given (in this case the device number of the paper tape punch is 12).

The following abbreviations are used exclusively in Section 3.1:

$k = e + c$ $\quad$ $y = w - d - 2$ $\quad$ $\bar{t} = \bar{w} - \bar{d} - 6$

$r = w - 1$ $\quad$ $t = w - d - 6$

$n|$ *FORMAT*(*Iw*)
PUNCH TAPE n,a

Knuth **output** *1*(device number,'$+rD$',*a*);

KDF9 *WRITE* (device number, *FORMAT*('$*D\cdot\cdot D$'),*a*); The number of *D*'s = r

NE *PRINT DIGITS*(*r*),*a*;

example:	printed result:
99| *FORMAT*(I5)	
PUNCH TAPE 99,A	77
output *1*(*12*,'$+4D$',*A*);	+0077
WRITE (*12*,*FORMAT*('$*DDDD$'),*A*);	+0077
PRINT DIGITS(*4*),*A*;	77

n| *FORMAT(Fw.d)*
PUNCH TAPE n,a

Knuth **output** *1*(device number,'+*yD.dD*',*a*);

KDF9 *WRITE* (device number, *FORMAT*('=*D··D.D··D*'),*a*);

The first group of *D*'s contains y characters and the second *d* characters.

NE *PRINT ALIGNED*(*y*,*d*),*a*;

example:	printed result:
98| *FORMAT*(*F12.4*)	938.2129
PUNCH TAPE 98,A	
output *1*(*12*,'+*6D.4D*',*A*);	+000938.2129
WRITE (*12*,*FORMAT*	+000938.2129
('=*DDDDDD.DDDD*'),*A*);	
PRINT ALIGNED(*6*,*4*),*A*;	938.2129

---

n| *FORMAT(Ew.d)*
PUNCH TAPE n,a

Knuth **output** *1*(device number,'+t*D.dD*$_{10}$+*2D*',*a*);

KDF9 *WRITE* (device number, *FORMAT*('=*D··D.D··D* @=*DD*'),*a*);

The first group of *D*'s contains t characters and the second *d* characters.

NE *PRINT SCALED*(*w* − *6*),*a*;

example:	printed result:
97| FORMAT (E14.5) *PUNCH TAPE 97,A*	6.64359E − 24
output *1(12,'+3D.5D$_{10}$ + 2D',A);*	+006.64359$_{10}$ − 24
WRITE (12,FORMAT *('*∗*DDD.DDDDD@*∗*DD'),A);*	+006.64359@ − 24
PRINT SCALED(8),A;	6.6435900@ − 24

---

n\| FORMAT(cHtext)
PUNCH TAPE n

Knuth **output** *1*(device number,"*text*");

KDF9 *WRITE TEXT* (device number, '*text*');

NE *PRINT* $*text?;*

example:	printed result:
96| FORMAT(10HHERE WE GO) *PUNCH TAPE 96*	HERE WE GO
output *1(12,"HERE WE GO");*	HERE WE GO
WRITE TEXT (12,'HERE WE GO');	HERE WE GO
PRINT $HERE WE GO?;	HERE WE GO

---

n| *FORMAT*({ },*cHtext*)
PUNCH TAPE n,a

Knuth **output** *1*(device number,'{ }'*text*'',*a*);

KDF9 *WRITE* (device number,*FORMAT*('{ }'),*a*);
WRITE TEXT (device number,'*text*');

NE *PRINT* { },*SAMELINE*,*a*,$ *text?*;

example:	printed result:
95| *FORMAT*(*I2*,*6HIS ODD*) *PUNCH TAPE 95*,*A*	3IS ODD
output *1*(*12*,'+*D*'*IS ODD*'',*A*);	+3IS ODD
WRITE (*12*,*FORMAT*('+*D*'),*A*);	+3IS ODD
WRITE TEXT (*12*,'*IS ODD*');	
PRINT DIGITS(*1*),*SAMELINE*,*A*,$*IS ODD?*;	3IS ODD

---

n| *FORMAT*({·},*eX*,{··})
PUNCH TAPE n,*a*,a_1

Knuth **output** *2*(device number, '{·}*eB*,{··}',*a*,a_1);

KDF9 *WRITE* (device number, *FORMAT*('{·};*eS*'),*a*);
WRITE (device number, *FORMAT*('{··}'),a_1);

NE *PRINT SAMELINE*,{·},*a*,$*Se?*,{··},$a_1$;

---

n| *FORMAT*(*Fw.d*//*E*$\overline{w}$.$\tilde{d}$)
PUNCH TAPE n,a,a$_1$

Knuth **output** *2*(device number, '+y*D.dD*//,+$\tilde{t}$*D.*$\tilde{d}$*D*$_{10}$ + *2D*',*a*,*a*$_1$);

KDF9 *WRITE* (device number,*FORMAT*('∗*D*··*D.D*··*D*;*CC*'),*a*);
WRITE (device number, *FORMAT*('∗*D*··*D.D*··*D* @ = *DD*'),*a*);

The first group of *D*'s contains y characters the second *d* characters the third $\tilde{t}$ and the fourth $\tilde{d}$ characters.

NE *PRINT ALIGNED*(*y,d*),*a*,$*L2?*,*SCALED*($\overline{w}$ − *6*),*a*$_1$;

example:	printed result:
93| *FORMAT*(*F7.2*//*E15.5*) *PUNCH TAPE 93,A,B*	137.04 ———— 2.42620*E* − 10
output *2*(*12*,'+*3D.2D*//,+*4D.5D*$_{10}$ + *2D*',*A*,*B*);	+137.04 ———— +0002.42620$_{10}$ − 10
WRITE(*12*,*FORMAT*('∗*DDD.DD*;*CC*'),*A*); *WRITE*(*12*,*FORMAT*('∗*DDDD.DDDDD*@∗*DD*'),*B*);	+137.04 ———— +0002.42620@ − 10
PRINT ALIGNED(*3,2*),*A*,$*L2?*, *SCALED*(*9*),*B*;	137.04 ———— 2.42620000@ − 10

n| *FORMAT*(*cHtext*{ })
PUNCH TAPE n,a

Knuth **output** *1*(device number,"*text*'{ }',*a*);

KDF9 *WRITE TEXT* (device number,'*text*');
WRITE (device number,*FORMAT*('{ }'),*a*);

NE *PRINT PREFIX*($*text?*),{ },*a*;

example: printed result:

92| *FORMAT*(*5H ETA = I4*) | ETA= −12 |
PUNCH TAPE 92,A
output *1*(*12*, "*ETA =*'+*3D*',*A*); | ETA= −012 |
WRITE TEXT (*12*,' *ETA =*'); | ETA= −012 |
WRITE (*12*,*FORMAT*(' + *DDD*'),*A*);
PRINT PREFIX($ *ETA = ?*), | ETA= −12 |
DIGITS(*3*),*A*;

---

n| *FORMAT*(*e*{·},*c*{··})
PUNCH TAPE n,$a_1, \ldots, a_e, a_{e+1}, \ldots, a_k$

Knuth **output** *k*(device number,'*e*({·}),*c*({··})',$a_1, \ldots, a_k$;

KDF9 *OUTPUT* (device number,$a_1, \ldots, a_k$);

NE *PRINT SAMELINE*,{·},$a_1, \ldots, a_e$,{··},$a_{e+1}, \ldots, a_k$;

example: printed result:

91| *FORMAT(2I5,3F10.3)*
PUNCH TAPE 91,A,B,C,D,E

22	1933	−7.532	0.010	22.222

output *5(12,'2(+4D),3(+5D.3D)',A,B,C,D,E);*

+0022 + 1933 − 00007.532 + 00000.010 + 00022.222

OUTPUT (12,A,B,C,D,E);

+2.20000000000@ + 01
+1.93300000000@ + 03
−7.53200000000@ + 00
+1.00000000000@ − 02
+2.22220000000@ + 01

PRINT SAMELINE,DIGITS(4),A,B,ALIGNED(5,3),C,D,E;

22	1933	−7.532	0.010	22.222

---

n| *FORMAT({ })*
WRITE(Nc,n)$a_1, \ldots, a_k$

Knuth **output** *k(Nc,'k({ })',*$a_1, \ldots, a_k$*);*

KDF9 OUTPUT (*Nc,*$a_1, \ldots, a_k$*);*

3.2 ALGOL PROCESSOR INPUT AND OUTPUT STATEMENTS AND THE APPROPRIATE FORTRAN INPUT AND OUTPUT STATEMENTS

In this section, input/output statements of different ALGOL processors are translated into FORTRAN II and FORTRAN IV. Only a few essential examples are chosen out of a possible great variety. Reference is made to sources of further details. As examples of the input/output device, paper tape reader and paper tape punch were chosen.

3.2.1 Knuth's Proposal*

The following statements are only a few examples of the great number of statements to manipulate input and output of data. All these statements are formulated as subprograms, only the call for these subprograms is given here.

input *1*(device number,' ',*a*); This statement causes the next number on the device mentioned to be read in and to be assigned to *a*. No format statement necessary (standard input).	*n*| *FORMAT*({ }) *ACCEPT TAPE n,a* The format must be set according to the data.
output *1*(device number,' ',*a*); (standard output)	*n*| *FORMAT*(*Ew.*ϵ) *PUNCH TAPE n,a* $w = \epsilon + 6$
output *1*(device number, '$+wD.dD$',*a*);†	*n*| *FORMAT*(*Fe.d*) *PUNCH TAPE n,a* $e = w + d + 2$
output *1*(device number, '$+rD$',*a*);†	*n*| *FORMAT*(*Iw*) *PUNCH TAPE n,a* $w = r + 1$
output *1*(device number, '$+tD.dD_{10} + 2D$',*a*);†	*n*| *FORMAT*(*Ew.d*) *PUNCH TAPE n,a* $w = t + d + 6$
output *1*(device number, "*text*");	*n*| *FORMAT*(*cHtext*) *PUNCH TAPE n*

* For more details see: A Proposal for Input/Output Convention in ALGOL 60, *Communs Ass. comput. Mach.* 7,273,628(1964).

† If there is no sign, then the data are assumed positive. If there is a negative sign, the negative data will appear with a minus sign, the sign will be suppressed if the data are positive.

3.2.2 English Electric-Leo Processor*

The following statements are only a few examples of the great variety of statements available to manipulate input and output of data. All these statements are formulated as library subprograms with a code number *ANc*. The necessary library subprograms for a particular program are placed in the outmost block of the program (that is, immediately after the first **begin**).

Example:

```
. . .
. . .
. . .
BEGIN
LIBRARY A1,A4,A12;
. . .
. . .
```

---

a: = *READ*(device number); | *n*| *FORMAT*({ })
ACCEPT TAPE n,a

This statement causes the next number on the device mentioned to be read in and to be assigned to *a*. No format statement necessary (standard input).

The format must be set according to the data.

OUTPUT (device number,*a*); | *n*| *FORMAT*(*E18.11*)
(standard output) | *PUNCH TAPE n,a*

WRITE (device number, | *n*| *FORMAT*(*Fe.d*)
FORMAT('−*D*··*D.D*··*D*');† | *PUNCH TAPE n,a*

The first group of *D*'s may contain *w* characters and the second *d* characters.

$e = w + d + 2$

---

* For more details see: *KDF9 ALGOL Users Manual*, English Electric-Leo Computer Ltd. (December 1964).

† If there is no sign, then the data are assumed positive. If there is a positive sign the appropriate sign of the data are always given out, left-hand zeros are suppressed. If there is a not equal (≠) sign the appropriate sign of the data are always given in the position specified by the ≠ sign.

WRITE (device number, *FORMAT*('−*D*··*D*');†	*n*|	*FORMAT*(*Iw*) *PUNCH TAPE n,a*
There may be r *D*'s.		$w = \mathrm{r} + 1$
WRITE (device number, *FORMAT*('−*D*··*D*.*D*··*D* @ − *DD*'),*a*);†	*n*|	*FORMAT*(*Ew.d*) *PUNCH TAPE n,a*
The first group of *D*'s may contain t characters and the second *d* characters.		$w = \mathrm{t} + d + 6$
WRITE TEXT (device number,'*text*');	*n*|	*FORMAT*(*cHtext*) *PUNCH TAPE n*

*3.2.3 Elliott Brothers Ltd. (NE) Processor**

General rule: The format setting applies to all values to the right of this setting up to the end of the output statement. If there are two conflicting format settings, the rightmost is valid.

READ $a_1, \ldots, a_k$;	*n*|	*FORMAT*({ }) *ACCEPT TAPE* $n, a_1, \ldots, a_k$
k values are read into memory and are correlated with the identifiers $a_1, \ldots, a_k$. No format statement necessary.		The format must be set according to the data.

* For more details see: 803 Library Program A104 ALGOL, Elliott Brothers Ltd.

PRINT $a_1, \ldots, a_k$;	n|	*FORMAT*(k*E*$w.\epsilon$) *PUNCH TAPE* $n,a_1, \ldots, a_k$
The k values of the identifiers $a_1, \ldots, a_k$ are printed out with ϵ significant digits times a power of ten for real numbers and with δ significant digits for integer numbers (standard output).	$w = \epsilon + 6$ or	
	n|	*FORMAT*(k*I*w) *PUNCH TAPE* $n,a_1, \ldots, a_k$
	$w = \delta + 1$	
PRINT DIGITS(r),a;	n|	*FORMAT*(*I*w) *PUNCH TAPE* n,a
	$w =$ r $+ 1$	
PRINT ALIGNED(w,d),a,	n|	*FORMAT*(*F*$e.d$) *PUNCH TAPE* n,a
	$e = w + d + 2$	
PRINT SCALED(e),a;	n|	*FORMAT*(*E*$w.d$) *PUNCH TAPE* n,a
	$w = e + 6$ $d = w - 7$	
PRINT $ *text* ?;	n|	*FORMAT*(c*H*text) *PUNCH TAPE* n

3.2.4 ALCOR Group*

READ($a_1, \ldots, a_k$);	n| *FORMAT*({ }) *ACCEPT TAPE* n,$a_1, \ldots, a_k$
k values are read into memory and are correlated with the identifiers $a_1, \ldots, a_k$. No format statement necessary.	The format must be set according to the data.
PRINT($a_1, \ldots, a_k$);	n| *FORMAT*(k*E*$w.\epsilon$) *PUNCH TAPE* n,$a_1, \ldots, a_k$
The k values of the identifiers $a_1, \ldots, a_k$ are printed out with ϵ significant digits times a power of ten for real numbers and with δ significant digits for integer numbers.	$w = \epsilon + 6$ or n| *FORMAT*(k*I*w) *PUNCH TAPE* n,$a_1, \ldots, a_k$ $w = \delta + 1$
WRITE("*text*");	n| *FORMAT*(c*H*text) *PUNCH TAPE* n

* For more details see: ALGOL-Manuel der ALCOR-Gruppe: *Elektronische Rechenanlagen* **5/6** (1961) and **2** (1962) and Appendix 5 of the manuals of the different ALCOR processors.

Part 4

ALGOL 60 → FORTRAN II and IV

On the following pages the left column contains ALGOL 60 statements and the right column the appropriate FORTRAN statements.* First the translation into FORTRAN II is given, this translation is also valid for FORTRAN IV, as long as no additional translation is given.† The beginning of a translation into FORTRAN IV is indicated by FIV.

The sequence of statements is in alphabetical order, except where this rule is broken to place together statements which belong together. After the translation of closely related statements (as **if . . . then** or **if . . . then . . . else** or **for . . . do**, etc.) a few examples of combinations of these statements are given. Some parts of a computer program (such as subprograms, **go to**, **switch**, etc.) are easier to translate when treated as a group of statements; such statements are grouped together. Groups with essential combinations of statements inside this group are translated first. After this, a few examples show how these groups can be combined together. Sometimes it is necessary to change the type of an identifier (as indicated by an *A* or *N* as the first character of this identifier), this change must be done in a preceding statement. The meaning of the notes (abbreviated by N) and restrictions (abbreviated by R) is given in Part 2.2 and 2.3. For explanation and example card reader and card punch are assumed as input/output devices. A reference to IBM 1620 includes both IBM 1620_{I} and 1620_{II}.

Important remark: Occasionally, the length of a FORTRAN statement exceeds the width of a line. To indicate that the next line should appear as a continuation of the previous line, we use two arrows → ←.

The following example makes this clear:

The statement

DIMENSION Amatrix(Nc_1),→
←*Amatrix*(Nc_2)

should actually be written as:

DIMENSION Amatrix(Nc_1),*Amatrix*(Nc_2)

* Statement numbers (labels) are separated from the statement itself by some spaces, indicating in this manner their actual appearance on a record.

† A translation into FORTRAN IV is given only if it is simpler than the FORTRAN II translation.

. . . *abs*(*Aa*) . . . ;	. . . *ABS*(*Aa*) . . .
	N1,N42 Exception: IBM 1401, IBM 1620_{I}, in this case: *IF*(*Aa*)n_1,n_2,n_2 n_1| *Aa* = −*Aa* n_2| . . . *Aa* . . .
. . . *abs*(*Na*) . . . ;	Aa = IABS(Na) . . . Aa . . .
	N1 Exception: IBM 1401, IBM 1620_{I}, in this case: *IF*(*Na*)n_1,n_2,n_2 n_1| *Aa* = −*Na* *GO TO* n_3 n_2| *Aa* = *Na* n_3| . . . *Aa* . . . Exception: IBM 1620_{II} *Aa* = *ABSF*(*Na*) . . . *Aa* . . .
. . . *abs*(*Aterm*) . . . ;	. . . *ABS*(*Aterm*) . . .
	N1,N2,N42 Exception: IBM 1401, IBM 1620_{I}, in this case: *Av* = *Aterm* *IF*(*Av*)n_1,n_2,n_2 n_1| *Av* = −*Av* n_2| . . . *Av* . . .

... *abs*(*Nterm*) ... ;	$Av = IABS(Nterm)$ ... Av ...
	N1,N2
	Exception: IBM 1401, IBM 1620_{I}, in this case:
	$Nv = Nterm$ $IF(Nv)n_1,n_2,n_2$ n_1 $Av = -Nv$ $GO\ TO\ n_3$ n_2 $Av = Nv$ n_3 ... Av ...
	Exception: IBM 1620_{II}:
	$Av = ABSF(Nterm)$ ... Av ...
addition:	
$a_1 + a_2$	$a_1 + a_2$
	N2
... *arctan*(*a*) ... ;	... $ATAN(Aa)$ ...
	R1,N42
... *arctan*(*term*) ... ;	... $ATAN(Aterm)$ ...
	R1,N2,N42
arithmetic operators:	
+	+
−	−
×	*
/	/
÷	not available (see under: division)
↑	**

array declaration:

see **array** *matrix*[ ] or **real array** *matrix*[ ] or **integer array** *matrix*[ ].

---

array *matrix*;

see also:

procedure *token*(. . , . . , *Nmatrix*, . . , . .);

This statement indicates that the parameter *matrix* of the subprogram is an array identifier.
Omit in translation.

---

array *matrix*[Nc_1:Nc_2];

DIMENSION Amatrix→
←(Nc_2)

R7
$Nc_2 \leqslant \beta$

---

array *matrix*[$-Nc_1$:Nc_2];

DIMENSION Amatrix→
←($Nc_1 + Nc_2 + 1$)

R7, N44
$Nc_1 + Nc_2 + 1 \leqslant \beta$

Whenever *Amatrix* appears in the program, the subscript must be increased by $N_1 + 1$.

example:

array *J*[−*12*:*38*];	*DIMENSION ZJ*(*51*)
. . .	. . .
. . .	. . .
. . .	. . .
J[*L*]: = *A* × *B*;	*ZJ*(*L* + *13*) = *A***B*

---

array *matrix*[$-Nc_1 : -Nc_2$]; *DIMENSION Amatrix*(Nc_1)

R7
$Nc_1 \leqslant \beta$
Whenever *Amatrix* appears in the program, one has to multiply the subscript by -1.
Since a negative subscript is not permitted, an additional statement must precede the appearance of the subscripted variable identifier.

example:

array *KURT*[$-30 : -10$];
...
...
...
KURT[*L*]: = *A*/*B*;

DIMENSION URT(*30*)
...
...
...
L1 = $-L$
URT(*L1*) = *A*/*B*

---

array *matrix*[*1:Nc*]; *DIMENSION Amatrix*(*Nc*)

R7
$Nc \leqslant \beta$

---

array *matrix*[*0:Nc*]; *DIMENSION Amatrix*→
←(*Nc* + *1*)

R7, N44
$Nc + 1 \leqslant \beta$

Whenever *Amatrix* appears in the program, the subscript must be increased by +1.

example:

array *DOOF*[*0:28*];
...
...
...
DOOF[*L*]: = *B*/*C* − *D*;

DIMENSION DOOF(*29*)
...
...
...
DOOF(*L* + *1*) = *B*/*C* − *D*

---

array *matrix*[**if** $a_1 < a_2$ **then** Nc_1 **else** $Nc_2 : Nc_3$];

DIMENSION Amatrix(Nc_3)

R7
$Nc_3 \leqslant \beta$

---

array *matrix*[Nc_1 : **if** $a \neq a_1$ **then** Nc_2 **else** Nc_3];

IF($a - a_1$)n,n_1,n
n| *DIMENSION Amatrix*(Nc_2)
GO TO n_2
n_1| *DIMENSION Amatrix*(Nc_3)
n_2| *CONTINUE*

R7
$Nc_2,Nc_3 \leqslant \beta$

---

array *matrix1,matrix2*[*1:Nc*];

DIMENSION Amatrix1→
←(*Nc*),*Amatrix2*(*Nc*)

R7
$Nc \leqslant \beta$

---

array *matrix*[Nc_1:Nc_2,Nc_3:Nc_4];

DIMENSION Amatrix→
←(Nc_2,Nc_4)

R7
$Nc_2,Nc_4 \leqslant \beta$

---

array *matrix*[*1:*Nc_1,*1:*Nc_2];

DIMENSION Amatrix→
←(Nc_1,Nc_2)

R7
$Nc_1,Nc_2 \leqslant \beta$

---

array *matrix*[Nc_1:Nc_2,Nc_3:Nc_4,
Nc_5:Nc_6];

DIMENSION Amatrix→
←(Nc_2,Nc_4,Nc_6)

R7
$Nc_2,Nc_4,Nc_6 \leqslant \beta$

---

array *matrix*[*1*:Nc_1,*1*:Nc_2,*1*:Nc_3];	*DIMENSION Amatrix*→ ←(Nc_1,Nc_2,Nc_3) R7 Nc_1,Nc_2,$Nc_3 \leqslant \beta$ Exceptions: IBM 1401 and IBM 1620_I permit a maximum of two dimensions only for an array.
array *matrix*[Nc_1:Nc_2,Nc_3: Nc_4, . . . , Nc_{k-1}:Nc_k];	For $k > 6$ no translation possible for IBM processors.
array *matrix1*,*matrix2*,*matrix3* [*1*:Nc_1],*matrix4*,*matrix5* [*1*:Nc_2,*1*:Nc_3];	*DIMENSION Amatrix1*→ ←(Nc_1),*Amatrix2*(Nc_1),→ ←*Amatrix3*(Nc_1),*Amatrix4*→ ←(Nc_2,Nc_3),*Amatrix5*→ ←(Nc_2,Nc_3) R7 Nc_1,Nc_2,$Nc_3 \leqslant \beta$
assignment statement:	
a_1: = a_2; a_2:N12	$a_1 = a_2$ a_2:N2,R3
logical assignment statement: $a = a_1$;	No logical assignment statement allowed in FORTRAN II. FIV: $a.EQ.a_1$
multiple assignment statement: a_1: = a_2: = a_3: = a_4; a_4:N12 All identifiers are of the same type.	$a_1 = a_4$ $a_2 = a_4$ $a_3 = a_4$

begin	Acts as an opening bracket. Omit in translation.
begin comment *text;* For the computation, this statement is equivalent to **begin**.	*C text* *text* must be written in columns 2 through 72, for a longer text additional cards must be used. These also must start with a *C* in column 1. No limit on the number of comment cards.
Boolean $v_1, \ldots, v_k$;	No logical variables allowed in FORTRAN II. FIV: *LOGICAL* $v_1, \ldots, v_k$
Boolean procedure *token;* . . . . **end**;	No logical subprogram allowed in FORTRAN II. FIV: *SUBROUTINE token* *COMMON* $v_i, \ldots, v_j$ . . . . *RETURN* *END* R5,N21,N29

Boolean procedure *token*(. . , . . , c_1, . . . , c_k, . . , . .);	No logical subprogram allowed in FORTRAN II.
. . .	
. . .	
. . .	FIV:
end;	*LOGICAL FUNCTION*→
	←*token*(. . , . . , c_1, . . . ,→
	←c_k, . . , . .)
N22	. . .
	. . .
	. . .
	RETURN
	END
	R5,N21,N29,N35

Boolean procedure *token*(. . . . . , v_1, . . . , v_k, . . , . .);	No logical subprogram allowed in FORTRAN II.
. . .	
. . .	
. . .	FIV:
end;	*LOGICAL FUNCTION*→
	←*token*(. . , . . , v_1, . . . ,→
	←v_k, . . , . .)
N22	*COMMON* v_i, . . . , v_j
	. . .
	. . .
	. . .
	RETURN
	END
	R5,N21,N29,N35

Boolean procedure *token*(. . , . . , *Amatrix*, . . , . .);	No logical subprogram allowed in FORTRAN II.
. . .	
array *Amatrix;*	
. . .	FIV:
. . .	*LOGICAL FUNCTION*→
. . .	←*token*(. . , . . , *Amatrix*,→
end;	←. . , . .)
	COMMON v_i, . . . , v_j
	DIMENSION Amatrix(*Nv*)
N22	. . .
	. . .
	. . .
	. . .
	RETURN
	END
	R5,N21,N29,N35
	Nv is an arbitrary variable identifier.

Boolean procedure *token*(. . , . . , *token1*, . . , . .);	No logical subprogram allowed in FORTRAN II.
. . .	
procedure token1;	
. . .	FIV:
. . .	*LOGICAL FUNCTION*→
. . .	←*token*(. . , . . , *token1*,→
end;	←. . , . .)
	COMMON v_i, . . . , v_j
	. . .
N22	
	. . .
	. . .
	. . .
	RETURN
	END

	R5,N21,N29,N35 The call for the subprogram must be preceded by *EXTERNAL→* *←token1*
Boolean procedure *token*(v,v_1,c) *text:*(a_1,v_2,a_2); . . . . **end**; N22	No logical subprograms allowed in FORTRAN II. FIV: *LOGICAL FUNCTION→* *←token*(v,v_1,c,a_1,v_2,a_2) *COMMON* $v_i, \ldots, v_j$ . . . . *RETURN* *END* R5,N21,N29,N35
Boolean procedure *token*(. . , . . . , . .)*text1:*(. . , . . . , . .)*text2:* (. . , . . . , . .); . . . . **end**; N22	*LOGICAL FUNCTION→* *←token*(. . , . . . , . . , . . ,→ ←. . . , . . , . . , . . . , . .) *COMMON* $v_i, \ldots, v_j$ . . . . *RETURN* *END* R5,N21,N29,N35
;**comment** *text;*	*C text* *text* must be written in columns 2 through 72, for a longer text additional cards must be used, also starting with a *C* in column 1. No limit on the number of comment cards.

. . . *cos*(*a*) . . . ;	. . . *COS*(*Aa*) . . .
	R1,N42
	FIV: . . . *COS*(*Aa*) . . .
	R1
. . . *cos*(*term*) . . . ;	. . . *COS*(*Aterm*) . . .
	R1,N2,N42
	FIV: . . . *COS*(*Aterm*) . . .
	R1,N2
division:	
a_1/a_2	Aa_1/Aa_2
a_1,a_2:N12	N2
The result is of real type.	
$Na_1 \div Na_2$	Na_1/Na_2
The arithmetic operator $\div$ is exclusively used to divide operands of the integer type.	

end;	*CONTINUE*
end	*END*
	end must be translated by *CONTINUE* if it is not the last (physical) statement of the ALGOL program. Exception: **end** in a **Boolean procedure** or **integer procedure** or **real procedure** or **procedure** must be translated as *RETURN* *END* If it is the last statement of the program then it must be translated by *END*.
end text;	*CONTINUE*
For the computation this statement is equivalent to **end**;.	*C* *text* or *END* *C* *text* See also under **end**. *text* must be written in columns 2 through 72, for a longer text additional cards must be used, also starting with a *C* in column 1. No limit on the number of comment cards.

. . . entier(a) . . . ;

```
      IF(a)n1,n2,n2
n1|   Nv = a - 1.
      GO TO n3
n2|   Nv = a
n3|   CONTINUE
      . . . Nv . . .
```

Note that the result is of integer type.

example:

A: = K + entier(C);

```
      IF(C)n1,n2,n2
n1|   NC = C - 1.
      GO TO n3
n2|   NC = C
n3|   CONTINUE
      A = K + NC
```

NC:N3

FIV:

. . . entier(Aa) . . . ;

```
      . . . AINT(Aa) . . .
```

. . . entier(Na) . . . ;

. . . entier(term) . . . ;

```
      IF(term)n1,n2,n2
n1|   Nv = term - 1.
      GO TO n3
n2|   Nv = term
n3|   CONTINUE
      . . . Nv . . .
```

term:N2

. . . entier(abs(a)) . . . ;

$Nv = ABS(a)$
$\ldots Nv \ldots$

Exception: IBM 1401, IBM 1620.
For IBM 1620_{II}:

$Nv = ABSF(a)$
$\ldots Nv \ldots$

example:

A: = K + entier(abs(C));

$NC = ABSF(C)$
$A = K + NC$

For IBM 1401 and IBM 1620_{I}:

$IF(a)n_1,n_2,n_2$
$n_1|$ $Nv = -a$
$GO\ TO\ n_3$
$n_2|$ $Nv = a$
$n_3|$ $CONTINUE$
$\ldots Nv \ldots$

example:

A: = L + entier(abs(C));

$IF(C)n_1,n_2,n_2$
$n_1|$ $NC = -C$
$GO\ TO\ n_3$
$n_2|$ $NC = C$
$n_3|$ $CONTINUE$
$A = L + NC$

. . . entier(abs(term)) . . . ;

$Nv = ABS(term)$
$\ldots Nv \ldots$

term :N2

Exception: IBM 1401, IBM 1620
For IBM 1620_{II}:

$Nv = ABSF(term)$
$\ldots Nv \ldots$

N2

For IBM 1401 and IBM 1620_{II}:

```
      IF(term)n1,n2,n2
n1|   Nv = -term
      GO TO n3
n2|   Nv = term
n3|   CONTINUE
      ... Nv ...
```

term :N2

... entier(arctan(a)) ... ;

```
Nv = ATAN(a)
... Nv ...
```

N42

Exception: IBM 1401:
Arctangent is not a predefined function, one has to evaluate the value by using a series approximation.

... entier(arctan(term)) ... ;

```
Nv = ATAN(term)
... Nv ...
```

term :N2
N42

Exception: IBM 1401:
Arctangent is not a predefined function, one has to evaluate the value by using a series approximation.

... entier(exp(a)) ... ;

```
Nv = EXP(a)
... Nv ...
```

N42

. . . *entier*(*exp*(*term*)) . . . ;	*Nv* = *EXP*(*term*) . . . *Nv* . . . *term* :N2 N42
. . . *entier*(*ln*(*a*)) . . . ;	*Nv* = *ALOG*(*a*) Exception: IBM 1401, IBM 1620, in this case: *Nv* = *LOGF*(*a*) . . . *Nv* . . .
. . . *entier*(*ln*(*term*)) . . . ;	*Nv* = *ALOG*(*term*) . . . *Nv* . . . *term* :N2 Exception: IBM 1401, IBM 1620, in this case: *Nv* = *LOGF*(*term*) . . . *Nv* . . . *term* :N2
. . . *entier*(*sqrt*(*a*)) . . . ;	*Nv* = *SQRT*(*a*) . . . *Nv* . . . N42 Exception: IBM 1401: *Nv* = *Aa***0.5* . . . *Nv* . . .

. . . *entier*(*sqrt*(*term*)) . . . ;	*Nv* = *SQRT*(*term*) . . . *Nv* . . .
	term :N2 N42
	Exception: IBM 1401: *Nv* = *Aterm****0.5* . . . *Nv* . . .
	Aterm :N2
. . . *exp*(*a*) . . . ;	. . . *EXP*(*Aa*) . . .
	R1,N42 FIV: . . . *EXP*(*Aa*) . . .
	R1
. . . *exp*(*term*) . . . ;	. . . *EXP*(*Aterm*) . . .
	R1,N2,N42 FIV: . . . *EXP*(*Aterm*) . . .
	R1,N2

exponentiation:

$a_1\uparrow a$	a_1**a

Exception:

$Na_1\uparrow Aa$	$Aa_1 = Na_1$ Aa_1**Aa
a,a_1 :N12	
example:	
$T := NC\uparrow AP;$	$AC = NC$ $T = AC$**AP

exponentiation:

$a_1\uparrow a_2\uparrow a$ $a_1\uparrow(a_2\uparrow a)$	$(a_1$**$a_2)$**a a_1**$(a_2$**$a)$
a,a_1,a_2 :N12	

expression:

The type of an expression will be integer if all the operands are integer, otherwise the expression is of real type.	The type of an expression will be integer if all the operands are integer. The type of an expression will be real if all the operands are real. If the operands of an expression are both of the integer type and of the real type, the expression is invalid.
. . . **false** . . . ;	No logical value allowed in FORTRAN II. FIV: . . . .*FALSE*. . .

for $v := a_1, a_2, \ldots, a_k$ **do**
begin
. . .
. . .
. . .
end;

$a_1, a_2, \ldots, a_k$:N12
N40

DO n Nv$_1$ *= 1,k*
GO TO($n_1, n_2, \ldots, n_k$),*Nv*$_1$
n_1| $v = a_1$
. . .
. . .
. . .
n_2| $v = a_2$

. . .
. . .
. . .
.
.
.
n_k| $v = a_k$
. . .
. . .
. . .
n| *CONTINUE*

N7
If . . . is very long, it is useful to
. . .
. . .
define the contents of . . . as a
. . .
. . .
SUBROUTINE. This *SUBROUTINE* may then be called instead of writing . . ..
. . .
. . .

R6

for $v := a_1$ **step** a_2 **until** a_3 **do**
begin
. . .

. . .
end;

a_1,a_2,a_3:N12
N40

```
      Na = ABS((a3 - a1)/→
     ←a2) + 1.0000001
      DO n Na7 = 1,Na
      a8 = Na7 - 1
      v = a8*a2 + a1
      . . .
      . . .
      . . .
n|    CONTINUE
```

N5,N7,N42,N43

Exception: IBM 1401, IBM 1620_{I}, in this case:

```
      Na = (a3 - a1)/a2
      IF(Na)n1,n1,n2
n1|   Na = -Na + 1
      GO TO n3
n2|   Na = Na + 1
n3|   DO n Na7 = 1,Na
      a8 = Na7 - 1
      v = a8*a2 + a1
      . . .
      . . .
      . . .
n|    CONTINUE
```

N5,N7

for *Nv*: = m_1 **step** m_2 **until** m_3 **do**	*DO n Nv* = m_1,m_3,m_2
begin	. . .
. . .	. . .
. . .	. . .
. . .	*n*| *CONTINUE*
end;	
	N7
N40	

for *Nv*: = *1* **step** *1* **until** m_3 **do**	*DO n Nv* = *1*,m_3
begin	. . .
. . .	. . .
. . .	. . .
. . .	*n*| *CONTINUE*
end;	
	N7
N40	

for *Nv*: = *0* **step** *1* **until** m_3 **do**	*Na* = m_3 + *1*
begin	*DO n* Na_7 = *1*,*Na*
. . .	*Nv* = Na_7 − *1*
. . .	. . .
. . .	. . .
end;	. . .
	n| *CONTINUE*
N40	
	N7

for $v: = a_1,a_2,a_3$ **step** a_4 **until** a_5, a_6 **do** . . . ;

$a_1, \ldots, a_6$:N12
N40

$v = a_1$
COMMON $v_i, \ldots, v_j$
CALL token(v)
$v = a_2$
CALL token(v)
$Na = ABS((a_5 - a_3)/\rightarrow$
$\leftarrow a_4) + 1.0000001$
DO n $Na_7 = 1,Na$
$a_8 = Na_7 - 1$
$v = a_8{*}a_4 + a_3$
CALL token(v)
$n|$ *CONTINUE*
$v = a_6$
CALL token(v)
CONTINUE
SUBROUTINE token(v)
COMMON $v_i, \ldots, v_j$
. . .
RETURN
END

R6,N5,N7,N21,N42,N43
Exception: IBM 1401, IBM 1620_I, in this case:

$v = a_1$
. . .
$v = a_2$
. . .
$Na = (a_5 - a_3)/a_4$
$IF(Na)n_1,n_1,n_2$
$n_1|$ $Na = -Na + 1$
GO TO n_3
$n_2|$ $Na = Na + 1$
$n_3|$ *DO n* $Na_7 = 1,Na$
$a_8 = Na_7 - 1$
$v = a_8{*}a_4 + a_3$
. . .
$v = a_6$
. . .
CONTINUE

N7

for *v*: = a_1,*term1* **while** $a_2 < a_3$
do . . . ;

a_1,a_2,a_3:N12
N40

```
      v = a1
      ...
n2|   v = term1
      IF(a2 - a3)n1,n,n
n1|   ...
      GO TO n2
 n|   CONTINUE
```

N7

---

for *v*: = a_1,*term1* **while** $a_2 \leqslant a_3$
do . . . ;

a_1,a_2,a_3:N12
N40

```
      v = a1
      ...
n2|   v = term1
      IF(a2 - a3)n1,n1,n
n1|   ...
      GO TO n2
 n|   CONTINUE
```

N7

---

for *a*: = a_1,*term1* **while** $a_2 = a_3$
do . . . ;

a_1,a_2,a_3:N12
N40

```
      v = a1
      ...
n2|   v = term1
      IF(a2 - a3)n,n1,n
n1|   ...
      GO TO n2
 n|   CONTINUE
```

N7

---

for *v*: = a_1,*term1* **while** $a_2 \geqslant a_3$
do . . . ;

a_1,a_2,a_3:N12
N40

```
      v = a1
      . . .
n2|   v = term1
      IF(a2 - a3)n,n1,n1
n1|   . . .
      GO TO n2
 n|   CONTINUE
```

N7

---

for *v*: = a_1,*term1* **while** $a_2 > a_3$
do . . . ;

a_1,a_2,a_3:N12
N40

```
      v = a1
      . . .
n2|   v = term1
      IF(a2 - a3)n,n,n1
n1|   . . .
      GO TO n2
 n|   CONTINUE
```

N7

---

for *v*: = a_1,*term1* **while** $a_2 \neq a_3$
do . . . ;

a_1,a_2,a_3:N12
N40

```
      v = a1
      . . .
n2|   v = term1
      IF(a2 - a3)n1,n,n1
n1|   . . .
      GO TO n2
 n|   CONTINUE
```

N7

---

for *v*: = *a*,*term1* **while** $a_1 = a_2 \equiv$		$v = a$
$a_3 > a_4$ **do** . . . ;		. . .
	n_4|	$v = term1$
		$IF(a_1 - a_2)n_2,n_1,n_2$
$a,a_1, \ldots, a_4$:N12	n_2|	$IF(a_3 - a_4)n_3,n_3,n$
N40	n_1|	$IF(a_3 - a_4)n,n,n_3$
	n_3|	. . .
		$GO\ TO\ n_4$
	n|	$CONTINUE$

for *v*: = *a*,*term1* **while** $a_1 = a_2$ V		$v = a$
$a_3 > a_4$ **do** . . . ;		. . .
	n_3|	$v = term1$
		$IF(a_1 - a_2)n,n_1,n$
$a,a_1, \ldots, a_4$:N12	n|	$IF(a_3 - a_4)n_2,n_2,n_1$
N40	n_1|	. . .
		$GO\ TO\ n_3$
	n_2|	$CONTINUE$

for *v*: = *a*,*term1* **while** $a_1 = a_2$ Λ		$v = a$
$a_3 > a_4$ **do** . . . ;		. . .
	n_3|	$v = term1$
		$IF(a_1 - a_2)n,n_1,n$
$a,a_1, \ldots, a_4$:N12	n_1|	$IF(a_3 - a_4)n,n,n_2$
N40	n_2|	. . .
		$GO\ TO\ n_3$
	n|	$CONTINUE$

for *v*: = *a*,*term1* **while** $a_1 = a_2 \supset$		$v = a$
$a_3 > a_4$ **do** . . . ;		. . .
	n_4|	$v = term1$
		$IF(a_1 - a_2)n_2,n_1,n_2$
$a,a_1, \ldots, a_4$:N12	n_2|	$IF(a_3 - a_4)n_3,n_3,n_1$
N40	n_1|	$IF(a_3 - a_4)n,n,n_3$
	n_3|	. . .
		$GO\ TO\ n_4$
	n|	$CONTINUE$

for *v*: = *a*,*term1* **while** ⊽ $a_1 > a_2$
do . . . ;

a,a_1,a_2:N12
N40

```
      v = a
      . . .
n2|   v = term1
      IF(a1 - a2)n1,n1,n
n1|   . . .
      GO TO n2
 n|   CONTINUE
```

---

for *a*: = a_1,*term1* **while** $a_2 \leqslant a_3$,a_4
do . . . ;

a_1, . . . , a_4:N12
N40

```
      v = a1
      COMMON vi, . . . , vj
      CALL token(v)
n2|   v = term1
      IF(a2 - a3)n1,n1,n
n1|   CALL token(v)
      GO TO n2
 n|   v = a4
      CALL token(v)
      CONTINUE
      SUBROUTINE token(v)
      COMMON vi, . . . , vj
      . . .
      RETURN
      END
```

R6,N7,N21

Exception: IBM 1401, IBM 1620_I, in this case:

```
      v = a1
      . . .
n2|   v = term1
      IF(a2 - a3)n1,n1,n
n1|   . . .
      GO TO n2
 n|   v = a4
      . . .
      CONTINUE
```

N7

for $v := a_1,a_2$ **step** a_3 **until** a_4,a_5, *term1* **while** $a_6 \neq a_7,a_8$ **do** . . . ;

$a_1, \ldots, a_8$:N12
N40

```
      v = a1
      COMMON vi, . . . , vj
      CALL token(v)
      Na = ABS((a4 − a2)/→
     ←a3) + 1.0000001
      DO n Na11 = 1,Na
      a12 = Na11 − 1
      v = a12*a3 + a2
      CALL token(v)
   n| CONTINUE
      v = a5
      CALL token(v)
  n3| v = term1
      IF(a6 − a7)n1,n2,n1
  n1| CALL token(v)
      GO TO n3
  n2| v = a8
      CALL token(v)
      CONTINUE
      SUBROUTINE token(v)
      COMMON vi, . . . , vj
      . . .
      RETURN
      END
```

R6,N5,N7,N21,N42,N43

Exception: IBM 1401, IBM 1620_I, in this case:

```
      v = a1
      . . .
      Na = (a4 − a2)/a3
```

```
      IF(Na)n1,n1,n2
n1|   Na = -Na + 1
      GO TO n3
n2|   Na = Na + 1
n3|   DO n Na11 = 1,Na
      a12 = Na11 - 1
      v = a12*a3 + a2
      . . .
n|    CONTINUE
      v = a5
      . . .
n6|   v = term1
      IF(a6 - a7)n4,n5,n4
n4|   . . .
      GO TO n6
n5|   v = a8
      . . .
      CONTINUE
```

N7

---

for $v := a_1$,*term1* **while** $a_2 > a_3$, a_4,a_5 **step** a_6 **until** a_7 **do** . . . ;

$a_1, \ldots, a_7$:N12
N40

```
      v = a1
      COMMON vi, . . . , vj
      CALL token(v)
      v = term1
n3|   IF(a2 - a3)n,n,n1
n1|   CALL token(v)
      GO TO n3
n|    v = a4
      CALL token(v)
      Na = ABS((a7 - a5)/→
      ←a6) + 1.0000001
      DO n2 Na11 = 1,Na
      a12 = Na11 - 1
      v = a12*a7 + a6
      CALL token(v)
n2|   CONTINUE
```

SUBROUTINE token(v)
COMMON $v_i, \ldots, v_j$
. . .
RETURN
END

R6,N5,N7,N21

Exception: IBM 1401, IBM 1620_I, in this case:

v = a_1
. . .
n_6| *v = term1*
IF($a_2 - a_3$)n,n,n_1
n_1| . . .
GO TO n_6
n| *v* = a_4
. . .
Na = ($a_7 - a_5$)/a_6
IF(*Na*)n_2,n_2,n_3
n_2| *Na* = −*Na* + *1*
GO TO n_4
n_3| *Na* = *Na* + *1*
n_4| *DO* n_5 Na_{11} = *1,Na*
$a_{12} = Na_{11} - 1$
v = a_{12}*a_6 + a_5
. . .
n_5| *CONTINUE*

N7

---

for *v*: = a_1,a_2 **step** a_3 **until** a_4,a_5,a_6 **step** a_7 **until** a_8,a_9 **do** . . . ;

$a_1, \ldots, a_9$:N12
N40

v = a_1
COMMON $v_i, \ldots, v_j$
CALL token(v)
Na = *ABS*(($a_4 - a_2$)/→
←a_3) + *1.0000001*
DO n Na_{11} = *1,Na*

```
      a12 = Na11 - 1
      v = a12*a3 + a2
      CALL token(v)
n|    CONTINUE
      v = a5
      CALL token(v)
      Na20 = ABS((a8 - a6)/→
      ←a7) + 1.0000001
      DO n1 Na21 = 1,Na20
      a22 = Na21 - 1
      v = a22*a7 + a6
      CALL token(v)
n1|   CONTINUE
      v = a9
      CALL token(v)
      CONTINUE
      SUBROUTINE token(v)
      COMMON vi, . . . , vj
      . . .
      RETURN
      END
```

R6,N5,N7,N21,N42,N43

Exception: IBM 1401, IBM 1620_{I}, in this case:

```
      v = a1
      . . .
      Na = (a4 - a2)/a3
      IF(Na)n2,n2,n3
n2|   Na = -Na - 1
      GO TO n4
n3|   Na = Na + 1
n4|   DO n5 Na11 = 1,Na
      a12 = Na11 - 1
      v = a12*a3 + a2
      . . .
n5|   CONTINUE
      v = a5
      . . .
      Na20 = (a8 - a6)/a7
```

```
      IF(Na20)n6,n6,n7
n6|   Na20 = -Na20 - 1
      GO TO n8
n7|   Na20 = Na20 + 1
n8|   DO n9 Na21 = 1,Na20
      a22 = Na21 - 1
      v = a22*a7 + a6
      . . .
n9|   CONTINUE
      v = a9
      . . .
      CONTINUE
```

N7

---

for *Nv*: = m_1 **step** m_2 **until** m_3 **do** input/output statement for: *matrix*[*Nv*];

```
n|    FORMAT({ })
      READ
      PUNCH n,(matrix(Nv),→
      ←Nv = m1,m3,m2)
```

R4,N8

Exception: IBM 1620_I, in this case:

```
n|    FORMAT({ })
      DO n1 Nv = m1,m3,m2
      READ
      PUNCH n, matrix(Nv)
n1|   CONTINUE
```

R4,N8

---

for *Nv*: = m_1 **step** $-m_2$ **until** m_3 **do** input/output statement for: *matrix*[*Nv*];

```
n|  FORMAT({ })
    Na = (m1 − m3)/m2 + 1
    DO n1 Na2 = 1,Na
    Nv = m1 + m2 − m2*Na2
    READ
    PUNCH n,matrix(Nv)
n1| CONTINUE
```

R4,N8

---

for *Nv*: = $-m_1$ **step** m_2 **until** m_3 **do** input/output statement for: *matrix*[*Nv*];

```
n|  FORMAT({ })
    m4 = m1 + 1
    READ
    PUNCH n,(matrix→
    ←(Nv + m4),Nv = 1,m3,m2)
```

R4,N8

See also: **array** *matrix*[$-Nc_1$:Nc_2]

example:

NE 803:

```
ARRAY Z(−2:10);
. . .
. . .
. . .
FOR K: = −2 STEP 2 UNTIL
10 DO READ Z(K);
. . .
. . .
THETA: = D*Z(K);
```

```
    DIMENSION Z(13)
    . . .
    . . .
    . . .
n|  FORMAT({ })
    READ n,(Z(K + 3),→
    ←K = 1,10,2
    . . .
    . . .
    THETA = D*Z(K + 3)
```

Exception: IBM 1620$_I$, in this case:

```
n|  FORMAT({ })
    m4 = m1 + 1
    DO n1 Nv = 1,m3,m2
    READ
    PUNCH n,matrix(Nv + m4)
n1| CONTINUE
```

R4,N8

for *Nv*: = $-m_1$ **step** $-m_2$ **until** $-m_3$ **do** input/output statement for:*matrix*[*Nv*];

```
n|  FORMAT({ })
    READ
    PUNCH n,(matrix(Nv),→
    ←Nv = m1,m3,m2)
```

R4,N8

See also: **array** *matrix*[$-Nc_1$: $-Nc_2$]

Exception: IBM 1620_I, in this case:

```
n|  FORMAT({ })
    DO n1 Nv = m1,m3,m2
    READ
    PUNCH n,matrix(Nv)
n1| CONTINUE
```

R4,N8

---

for *Nv*: = m_1 **step** *1* **until** m_2 **do** input/output statement for: *matrix*[*Nv*];

```
n|  FORMAT({ })
    READ
    PUNCH n,(matrix(Nv),→
    ←Nv = m1,m2)
```

R4,N8

Exception: IBM 1620_I, in this case:

```
n|  FORMAT({ })
    DO n1 Nv = m1,m2
    READ
    PUNCH n,matrix(Nv)
n1| CONTINUE
```

R4,N8

for *Nv*: = *1* **step** *1* **until** m_3 **do** input/output statement for *matrix*[*Nv*];

```
n|  FORMAT({ })
    READ
    PUNCH n,(matrix(Nv), →
    ←Nv = 1,m3)
```

R4,N8

Exception: IBM 1620_I, in this case:

```
 n|  FORMAT({ })
     DO n1 Nv = 1,m3
     READ
     PUNCH n,matrix(Nv)
n1|  CONTINUE
```

R4,N8

---

for *Nv*: = *0* **step** m_2 **until** m_3 **do** input/output statement for: *matrix*[*Nv*];

```
n|  FORMAT({ })
    READ
    PUNCH n,(matrix(Nv + →
    ←1),Nv = 1,m3,m2)
```

R4,N8

See also: **array** *matrix*[*0* :*Nc*]
Exception: IBM 1620_I: in this case:

```
 n|  FORMAT({ })
     DO n1 Nv = 1,m3,m2
     READ
     PUNCH n,matrix(Nv + 1)
n1|  CONTINUE
```

R4,N8

---

for *Nv: = 1* **step** *1* **until** m_3 **do** input/output statement for: *matrix*[*Nv*],*matrix1*[*Nv*];

```
n|  FORMAT({ },{ })
    READ
    PUNCH n,(matrix(Nv),→
    ←matrix1(Nv),Nv = 1,m3)
```

R4,N8

Exception: IBM 1620_I, in this case:

```
n|  FORMAT({ },{ })
    DO n1 Nv = 1,m3
    READ
    PUNCH n,matrix(Nv),→
    ←matrix1(Nv)
n1| CONTINUE
```

R4,N8

---

go to *n;*

GO TO n

N20
Leading zeros do not affect the meaning of the label, i.e.:
go to *0015;* is equivalent to **go to** *15;*

$n \leqslant \zeta$

---

go to *v;* . . . . *v:*		*ASSIGN n TO v* *GO TO v,(n)* . . . . *n*| . . . . . .
N20	$n \leqslant \zeta$ Exception: IBM 1401, IBM 1620, in this case: *v* must be replaced by an integer number *n*.	*GO TO n* . . . . *n*| . . . . . .
go to *jump*[*Nv*];		*GO TO*(n_1,n_2, . . . , n_k),*Nv*
N19,N33	$Nv \leqslant 10$	
go to *jump*[*term*];		$Nv = term$ *GO TO*(n_1,n_2, . . . , n_k),*Nv*
N19,N33	$Nv \leqslant 10$ *term* :N2	

---

go to *jump*[**if** $v < a$ **then** Nc_1 **else** Nc_2];

a:N12

N19,N33

```
      IF(v - a)n_n,n_m,n_m
n_n|  Nv_1 = Nc_1
      GO TO n
n_m|  Nv_1 = Nc_2
  n|  GO TO(n_1,n_2, . . . , n_k),Nv_1
```

$Nv_1 \leqslant 10$

$n_n, n_m \neq n_1, n_2, \ldots, n_k$

---

go to *jump*[**if** $v \leqslant a$ **then** Nc_1 **else** Nc_2];

a:N12

N19,N33

```
      IF(v - a)n_n,n_m,n_m
n_n|  Nv_1 = Nc_1
      GO TO n
n_m|  Nv_1 = Nc_2
  n|  GO TO (n_1,n_2, . . . , n_k),Nv_1
```

$Nv_1 \leqslant 10$

$n_n, n_m \neq n_1, n_2, \ldots, n_k$

---

go to *jump*[**if** $v = a$ **then** Nc_1 **else** Nc_2];

a:N12

N19,N33

```
      IF(v - a)n_n,n_m,n_m
n_n|  Nv_1 = Nc_2
      GO TO n
n_m|  Nv_1 = Nc_1
  n|  GO TO(n_1,n_2, . . . , n_k),Nv_1
```

$Nv_1 \leqslant 10$

$n_n, n_m \neq n_1, n_2, \ldots, n_k$

---

go to *jump*[**if** $v \geqslant a$ **then** Nc_1 **else** Nc_2];

a:N12

N19,N33

```
      IF(v - a)n_n,n_m,n_m
n_n|  Nv_1 = Nc_2
      GO TO n
n_m|  Nv_1 = Nc_1
  n|  GO TO(n_1,n_2, . . . , n_k),Nv_1
```

$Nv_1 \leqslant 10$

$n_n, n_m \neq n_1, n_2, \ldots, n_k$

---

go to *jump*[**if** $v > a$ **then** Nc_1 **else** Nc_2];		$IF(v - a)n_n,n_n,n_m$
	n_n	$Nv_1 = Nc_2$
		$GO\ TO\ n$
	n_m	$Nv_1 = Nc_1$
a:N12	n	$GO\ TO\ (n_1,n_2, \ldots, n_k),Nv_1$
N19,N33		$Nv_1 \leqslant 10$
		$n_n,n_m \neq n_1,n_2, \ldots, n_k$

go to *jump*[**if** $v \neq a$ **then** Nc_1 **else** Nc_2];		$IF(v - a)n_n,n_m,n_n$
	n_n	$Nv_1 = Nc_1$
		$GO\ TO\ n$
	n_m	$Nv_1 = Nc_2$
a:N12	n	$GO\ TO(n_1,n_2, \ldots, n_k),Nv_1$
N19,N33		$Nv_1 \leqslant 10$
		$n_n,n_m \neq n_1,n_2, \ldots, n_k$

go to if $v < a$ **then** Nc_1 **else** Nc_2;		$IF(v - a)n_1,n_2,n_2$
	n_1	$GO\ TO\ Nc_1$
	n_2	$GO\ TO\ Nc_2$
a:N12		CONTINUE
N20		
		$Nc_1,Nc_2 \leqslant \zeta$

go to if $v \leqslant a$ **then** Nc_1 **else** Nc_2;		$IF(v - a)n_1,n_1,n_2$
	n_1	$GO\ TO\ Nc_1$
	n_2	$GO\ TO\ Nc_2$
a:N12		$CONTINUE$
N20		$Nc_1,Nc_2 \leqslant \zeta$

go to if $v = a$ **then** Nc_1 **else** Nc_2;		$IF(v - a)n_1,n_2,n_1$
	n_1	$GO\ TO\ Nc_2$
	n_2	$GO\ TO\ Nc_1$
a:N12		$CONTINUE$
N20		$Nc_1,Nc_2 \leqslant \zeta$

go to if $v \geqslant a$ **then** Nc_1 **else** Nc_2;		$IF(v - a)n_1,n_2,n_2$
	n_1	$GO\ TO\ Nc_2$
	n_2	$GO\ TO\ Nc_1$
a:N12		$CONTINUE$
N20		$Nc_1,Nc_2 \leqslant \zeta$

go to if $v > a$ **then** Nc_1 **else** Nc_2;	$IF(v - a)n_1,n_1,n_2$
	n_1| $GO\ TO\ Nc_2$
	n_2| $GO\ TO\ Nc_1$
a:N12	$CONTINUE$
N20	$Nc_1,Nc_2 \leqslant \zeta$

go to if $v \neq a$ **then** Nc_1 **else** Nc_2;	$IF(v - a)n_1,n_2,n_1$
	n_1| $GO\ TO\ Nc_1$
	n_2| $GO\ TO\ Nc_2$
a:N12	$CONTINUE$
N20	$Nc_1,Nc_2 \leqslant \zeta$

go to if $v < a$ **then** Nc_1 **else** *jump* $[Nv_1]$;	$IF(v - a)n_n,n_m,n_m$
	n_n| $GO\ TO\ Nc_1$
	n_m| $GO\ TO(n_1,n_2, \ldots, n_k),Nv_1$
	$CONTINUE$
a:N12	$Nv_1 \leqslant 10$
N19,N20,N33	$n_n,n_m \neq n_1,n_2, \ldots, n_k$

if $a_1 < a$ **then** . . . ;	$IF(a_1 - a)n_1,n,n$
	n_1| . . .
	n| $CONTINUE$
a,a_1:N12	FIV:
	$IF(a_1.LT.a)$ . . .
	N34,N39

if $a_1 \leqslant a$ **then** . . . ;	$IF(a_1 - a)n_1,n_1,n$
	n_1| . . .
	n| $CONTINUE$
a,a_1:N12	FIV:
	$IF(a_1.LE.a)$ . . .
	N34,N39

if $a_1 = a$ **then** . . . ;	$IF(a_1 - a)n,n_1,n$
	n_1| . . .
	n| $CONTINUE$
a,a_1:N12	FIV:
	$IF(a_1.EQ.a)$ . . .
	N34,N39

if $a_1 \geqslant a$ **then** . . . ;	$IF(a_1 - a)n,n_1,n_1$
	n_1| . . .
	n| $CONTINUE$
a,a_1:N12	FIV:
	$IF(a_1.GE.a)$ . . .
	N34,N39

if $a_1 > a$ **then** . . . ;	$IF(a_1 - a)n,n,n_1$
	n_1| . . .
	n| $CONTINUE$
a,a_1:N12	FIV:
	$IF(a_1.GT.a)$ . . .
	N34,N39

if $a_1 \neq a$ **then** . . . ;	$IF(a_1 - a)n_1,n,n_1$
	n_1| . . .
	n| $CONTINUE$
a,a_1:N12	FIV:
	$IF(a_1.NE.a)$ . . .
	N34,N39

. . .
if $a_1 > a$ **then** v: . . . ;

a,a_1:N12

```
      IF(a1 - a)n,n,n1
n1|   CONTINUE
      . . .
nk|   . . .
 n|   CONTINUE
```

The label v must be replaced by the integer number n_k.

FIV:

```
      IF(a1.GT.a) GO TO n
      GO TO n1
 n|   CONTINUE
      . . .
nk|   . . .
n1|   CONTINUE
```

N34

$v: = a +$ (**if** $a_1 \neq a_2$ **then** a_3);

a,a_1,a_2,a_3:N12

```
      IF(a1 - a2)n1,n,n1
n1|   v = a + a3
      GO TO n2
 n|   v = a
n2|   CONTINUE
```

FIV:

```
      IF(a1.NE.a2)v = a + a3
      IF(.NOT.(a1.NE.a2))v = a
```

N34

$v: = a + ($**if** $a_1 \neq a_2$ **then** $a_3) + a_4;$		$IF(a_1 - a_2)n_1,n_2,n_1$
	n_1|	$v = a + a_3 + a_4$
		$GO\ TO\ n$
$a,a_1, \ldots, a_4$:N12	n_2|	$v = a + a_4$
	n|	$CONTINUE$
	FIV:	
		$IF(a_1.NE.a_2)v = a + a_3 + a_4$
		$IF(.NOT.(a_1.NE.a_2))\rightarrow$
		$\leftarrow v = a + a_4$
	N34	

		. . .
. . .		. . .
if $a_1 < a$ **then** . . . **else** . . . ;		$IF(a_1 - a)n_1,n_2,n_2$
	n_1|	. . .
		. . .
a,a_1:N12		$GO\ TO\ n$
	n_2|	. . .
		. . .
		. . .
	n|	$CONTINUE$
	FIV:	
		. . .
		$IF(a_1.LT.a)$ . . .
		. . .
		. . .
		. . .
	N34,N38	

```
                    . . .
               . . .    . . .
if a1 ≤ a then . . . else . . . ;        IF(a1 - a)n1,n1,n2
                                    n1| . . .
                                      . . .
a,a1:N12                              GO TO n
                                    n2| . . .
                                      . . .
                                      . . .
                                     n| CONTINUE
                                    FIV:
                                                  . . .
                                      IF(a1.LE.a) . . .
                                      . . .
                                      . . .
                                      . . .
```

N34,N38

```
                    . . .
               . . .    . . .
if a1 = a then . . . else . . . ;        IF(a1 - a)n1,n2,n1
                                    n2| . . .
                                      . . .
a,a1:N12                              GO TO n
                                    n1| . . .
                                      . . .
                                      . . .
                                     n| CONTINUE
                                    FIV:
                                                  . . .
                                      IF(a1.EQ.a) . . .
                                      . . .
                                      . . .
                                      . . .
```

N34,N38

```
                    . . .
              . . .     . . .
if a1 ⩾ a then . . . else . . . ;        IF(a1 - a)n1,n2,n2
                                    n2| . . .
                                      . . .
                                        GO TO n
                                      . . .
a,a1:N12                            n1| . . .
                                      . . .
                                     n| CONTINUE
                                    FIV:
                                                  . . .
                                        IF(a1.GE.a) . . .
                                      . . .
                                      . . .
                                      . . .
```

N34,N38

```
                    . . .
              . . .     . . .
if a1 > a then . . . else . . . ;        IF(a1 - a)n1,n1,n2
                                    n2| . . .
                                      . . .
a,a1:N12                                GO TO n
                                    n1| . . .
                                      . . .
                                      . . .
                                     n| CONTINUE
                                    FIV:
                                                  . . .
                                        IF(a1.GT.a) . . .
                                      . . .
                                      . . .
                                      . . .
```

N34,N38

```
                    . . .
               . . .    . . .
if a1 ≠ a then . . . else . . . ;
```

a,a1:N12

```
      IF(a1 - a)n1,n2,n1
n1|   . . .
      . . .
      GO TO n
n2|   . . .
      . . .
      . . .
 n|   CONTINUE
FIV:
                          . . .
      IF(a1.NE.a) . . .
      . . .
      . . .
      . . .
```

N34,N38

```
                    . . .
               . . .    . . .
if a1 ≤ a then v: . . . else . . . ;
```

a,a1:N12

```
      IF(a1 - a)n2,n2,n1
n2|   CONTINUE
nk|   . . .
      . . .
      GO TO n
n1|   . . .
      . . .
      . . .
 n|   CONTINUE
```

The label v must be replaced by the integer number n_k.

```
FIV:
      IF(a1.LE.a) GO TO n1
      . . .
      . . .
      . . .
      GO TO n
```

```
      n1| CONTINUE
      nk| . . .
        . . .
       n| CONTINUE
```

N34

if $a_1 = a_2$ **then begin if** $a_3 > a_4$
. . . . . .
then . . . else . . . end;
. . .

$a_1, \ldots, a_4$:N12

```
        IF(a1 - a2)n,n1,n
      n1| IF(a3 - a4)n3,n3,n4
      n4| . . .
        . . .
        GO TO n
      n3| . . .
        . . .
        . . .
       n| CONTINUE
FIV:
        IF(a1.EQ.a2) GO TO n1
        GO TO n
                        . . .
      n1| IF(a3.GT.a4) . . .
        . . .
        . . .
        . . .
       n| CONTINUE
```

N34,N38

if $a_1 \geqslant a_2$ **then begin if** $a_3 < a_4$
. . .
then . . . else . . . end else if $a_5 = a_6$
. . .
then . . . ;
. . .

$a_1, \ldots, a_6$:N12

```
        IF(a1 - a2)n1,n2,n2
      n2| IF(a3 - a4)n3,n4,n4
      n3| . . .
        GO TO n
      n4| . . .
        . . .
        GO TO n
      n1| IF(a5 - a6)n,n5,n
      n5| . . .
        . . .
        . . .
       n| CONTINUE
```

```
      FIV:
            IF(a1.GE.a2) GO TO n1
            GO TO n2
      n1|   IF(a3.LT.a4) . . .
            . . .
            . . .
            GO TO n
                                 . . .
      n2|   IF(a5.EQ.a6) . . .
                                 . . .
       n|   CONTINUE
```

N34,N38,N39

$v := a + ($**if** $a_1 < a_2$ **then** a_3 **else** $a_4)$;

$a,a_1, \ldots, a_4$:N12

```
            IF(a1 - a2)n1,n2,n2
      n1|   v = a + a3
            GO TO n
      n2|   v = a + a4
       n|   CONTINUE
      FIV:
            IF(a1.LT.a2) v = a + a3
            IF(.NOT.(a1.LT.a2))→
            ←v = a + a4
```

N34

$v := a + ($**if** $a_1 < a_2$ **then** a_3 **else** $a_4) + a_5$;

$a,a_1, \ldots, a_5$:N12

```
            IF(a1 - a2)n1,n2,n2
      n1|   v = a + a3 + a5
            GO TO n
      n2|   v = a + a4 + a5
       n|   CONTINUE
      FIV:
            IF(a1.LT.a2)v = a + a3→
            ←+ a5
            IF(.NOT.(a1.LT.a2))→
            ←v = a + a4 + a5
```

N34

---

if $a < a_1 \Lambda a_2 < a_3$ **then** . . . ;

a,a_1,a_2,a_3:N12

```
      IF(a - a1)n1,n,n
n1|   IF(a2 - a3)n2,n,n
n2|   . . .
 n|   CONTINUE
FIV:
      IF(a.LT.a1.AND.a2.→
      ←LT.a3) . . .
```

N34,N39

---

if $a < a_1 \Lambda a_2 \leqslant a_3$ **then** . . . ;

a,a_1,a_2,a_3:N12

```
      IF(a - a1)n1,n,n
n1|   IF(a2 - a3)n2,n2,n
n2|   . . .
 n|   CONTINUE
FIV:
      IF(a.LT.a1.AND.a2.→
      ←LE.a3) . . .
```

N34,N39

---

if $a < a_1 \Lambda a_2 = a_3$ **then** . . . ;

a,a_1,a_2,a_3:N12

```
      IF(a - a1)n1,n,n
n1|   IF(a2 - a3)n,n2,n
n2|   . . .
 n|   CONTINUE
FIV:
      IF(a.LT.a1.AND.a2.→
      ←EQ.a3) . . .
```

N34,N39

---

if $a < a_1 \Lambda a_2 \geqslant a_3$ **then** . . . ;

a,a_1,a_2,a_3:N12

```
      IF(a - a1)n1,n,n
n1|   IF(a2 - a3)n,n2,n2
n2|   . . .
 n|   CONTINUE
```

```
                        FIV:
                              IF(a.LT.a1.AND.a2.→
                              ←GE.a3) . . .
```

N34,N39

---

if $a < a_1 \Lambda a_2 \neq a_3$ **then** . . . ;

a,a_1,a_2,a_3:N12

```
                              IF(a - a1)n1,n,n
                        n1|   IF(a2 - a3)n2,n,n2
                        n2|   . . .
                         n|   CONTINUE
                        FIV:
                              IF(a.LT.a1.AND.a2.→
                              ←NE.a3) . . .
```

N34,N39

---

if $a < a_1 \Lambda a_2 < a_3$ **then** . . . **else**
. . .
. . . ;

a,a_1,a_2,a_3:N12

```
                              IF(a - a1)n1,n,n
                        n1|   IF(a2 - a3)n2,n,n
                        n2|   . . .
                              GO TO n3
                         n|   . . .
                              . . .
                        n3|   CONTINUE
                        FIV:
                              IF(a.LT.a1.AND.a2.→
                              ←LT.a3) . . .
                              IF(.NOT.(a.LT.a1.AND.→
                                       . . .
                              ←a2.LT.a3)) . . .
```

N34,N37

---

if $a < a_1 \ V \ a_2 < a_3$ **then** . . . ;

a,a_1,a_2,a_3:N12

```
                              IF(a - a1)n1,n2,n2
                        n2|   IF(a2 - a3)n1,n,n
                        n1|   . . .
                         n|   CONTINUE
```

FIV:
IF(a.LT.a₁.OR.a₂.LT.→
←a₃) . . .

N34,N39

---

if $a < a_1\ V\ a_2 \leqslant a_3$ **then . . . ;**

a,a_1,a_2,a_3:N12

IF$(a - a_1)n_1,n_2,n_2$
n_2| *IF*$(a_2 - a_3)n_1,n_1,n$
n_1| . . .
n| *CONTINUE*
FIV:
IF(a.LT.a₁.OR.a₂.LE.→
←a₃) . . .

N34,N39

---

if $a < a_1\ V\ a_2 = a_3$ **then . . . ;**

a,a_1,a_2,a_3:N12

IF$(a - a_1)n_1,n_2,n_2$
n_2| *IF*$(a_2 - a_3)n,n_1,n$
n_1| . . .
n| *CONTINUE*
FIV:
IF(a.LT.a₁.OR.a₂.EQ.→
←a₃) . . .

N34,N39

---

if $a < a_1\ V\ a_2 > a_3$ **then . . . ;**

a,a_1,a_2,a_3:N12

IF$(a - a_1)n_1,n,n$
n_1| *IF*$(a_2 - a_3)n,n,n_2$
n_2| . . .
n| *CONTINUE*
FIV:
IF(a.LT.a₁.AND.a₂.→
←GT.a₃) . . .

N34,N39

---

if $a < a_1\ V\ a_2 \geqslant a_3$ **then** . . . ;

a,a_1,a_2,a_3:N12

```
      IF(a - a1)n1,n2,n2
n2|   IF(a2 - a3)n,n1,n1
n1|   . . .
 n|   CONTINUE
FIV:
      IF(a.LT.a1.OR.a2.GE.→
      ←a3) . . .
```

N34,N39

---

if $a < a_1\ V\ a_2 > a_3$ **then** . . . ;

a,a_1,a_2,a_3:N12

```
      IF(a - a1)n1,n2,n2
n2|   IF(a2 - a3)n,n,n1
n1|   . . .
 n|   CONTINUE
FIV:
      IF(a.LT.a1.OR.a2.GT.a3) . . .
```

N34,N39

---

if $a < a_1\ V\ a_2 \neq a_3$ **then** . . . ;

a,a_1,a_2,a_3:N12

```
      IF(a - a1)n1,n2,n2
n2|   IF(a2 - a3)n1,n,n1
n1|   . . .
 n|   CONTINUE
FIV:
      IF(a.LT.a1.OR.a2.NE.→
      ←a3) . . .
```

N34,N39

---

if $a < a_1\ V\ a_2 < a_3$ **then** . . . **else** . . . ;

a,a_1,a_2,a_3:N12

```
      IF(a - a1)n1,n2,n2
n2|   IF(a2 - a3)n1,n,n
n1|   . . .
      GO TO n3
 n|   . . .
      . . .
n3|   CONTINUE
```

FIV:
$IF(a.LT.a_1.OR.a_2.LT.\rightarrow$
$\leftarrow a_3) \ldots$
$IF(.NOT.(a.LT.a_1.OR.\rightarrow$
. . .
$\leftarrow a_2.LT.a_3)) \ldots$

N34,N37

---

if ▽ $a \neq a_1$ **then** . . . ;

$IF(a - a_1)n,n_1,n$
$n_1|$ . . .
$n|$ $CONTINUE$

a,a_1:N12

FIV:
$IF(.NOT.(a.NE.a_1)) \ldots$

N34,N39

The logical expression to which *.NOT.* applies must be enclosed in parentheses if it contains two or more quantities.

. . .
if ▽ $a \neq a_1$ **then** . . . **else** . . . ;

$IF(a - a_1)n,n_1,n$
$n_1|$ . . .
$GO\ TO\ n_2$

a,a_1:N12

$n|$ . . .
. . .
$n_2|$ $CONTINUE$

FIV:
$IF(.NOT.(a.NE.a_1)) \ldots$
. . .
$IF(a.NE.a_1) \ldots$

N34,N37

The logical expression to which *.NOT.* applies must be enclosed in parentheses if it contains two or more quantities.

---

if $a < a_1 \equiv a_2 = a_3$ **then** . . . ;

a,a_1,a_2,a_3:N12

```
      IF(a - a1)n1,n2,n2
n2|   IF(a2 - a3)n,n3,n
n1|   IF(a2 - a3)n3,n,n3
n3|   . . .
 n|   CONTINUE
FIV:
      IF(a.LT.a1) GO TO n1
      IF(a2.NE.a3) GO TO n
n2|   . . .
      GO TO n
n1|   IF(a2.NE.a3) GO TO n2
 n|   CONTINUE
```

N34

---

if $a < a_1 \equiv a_2 \neq a_3$ **then** . . . **else**
. . .
. . . ;

a,a_1,a_2,a_3:N12

```
      IF(a - a1)n1,n2,n2
n2|   IF(a2 - a3)n,n3,n
n1|   IF(a2 - a3)n3,n,n3
n3|   . . .
      GO TO n4
 n|   . . .
      . . .
n4|   CONTINUE
FIV:
      IF(a.LT.a1) GO TO n1
      IF(a2.NE.a3) GO TO n
n2|   . . .
      GO TO n3
n1|   IF(a2.NE.a3) GO TO n2
 n|   . . .
      . . .
n3|   CONTINUE
```

N34

---

if $a < a_1 \supset a_2 \neq a_3$ **then** . . . ;

a,a_1,a_2,a_3:N12

```
      IF(a - a1)n1,n2,n2
n2|   IF(a2 - a3)n1,n3,n1
n1|   IF(a2 - a3)n3,n,n3
n3|   . . .
 n|   CONTINUE
FIV:
      IF(a.LT.a1) GO TO n1
      IF(a2.NE.a3) GO TO n1
n2|   . . .
      GO TO n
n1|   IF(a2.NE.a3) GO TO n2
 n|   CONTINUE
```

N34

---

if $a < a_1 \supset a_2 \neq a_3$ **then** . . . **else** . . . ;

a,a_1,a_2,a_3:N12

```
      IF(a - a1)n1,n2,n2
n2|   IF(a2 - a3)n1,n3,n1
n1|   IF(a2 - a3)n3,n,n3
n3|   . . .
      GO TO n4
 n|   . . .
      . . .
n4|   CONTINUE
FIV:
      IF(a.LT.a1) GO TO n1
      IF(a2.NE.a3) GO TO n1
n2|   . . .
      GO TO n3
n1|   IF(a2.NE.a3) GO TO n2
 n|   . . .
      . . .
n3|   CONTINUE
```

N34

---

integer $v_1,v_2, \ldots, v_k$;

No type declaration necessary. The variable identifiers must be changed into $Nv_1,Nv_2, \ldots, Nv_k$ throughout the program.
R2
FIV:
INTEGER $v_1,v_2, \ldots, v_k$

N30

---

integer array *matrix;*
see also:
procedure *token*(*Nmatrix*);

This statement indicates that the parameter *matrix* of the subprogram is an array.
Omit in translation.

---

integer array *matrix*[Nc_1:Nc_2];

DIMENSION Nmatrix→
←(Nc_2)

R7
$Nc_2 \leqslant \beta$

---

integer array matrix[$-Nc_1$:Nc_2];

DIMENSION Nmatrix→
←($Nc_1 + Nc_2 + 1$)

R7,N44
$Nc_1 + Nc_2 + 1 \leqslant \beta$

Whenever *Nmatrix* appears in the program, the subscript must be increased by $Nc_1 + 1$.

example:

integer array *Z*[−*13:47*];
...
...
...
Z[*L*]: = *N*/*M*;

DIMENSION NZ(*61*)
...
...
...
NZ(*L* + *14*) = *N*/*M*

---

integer array *matrix*[$-Nc_1$:$-Nc_2$]; *DIMENSION Nmatrix*→
←(Nc_1)

R7
$Nc_1 \leqslant \beta$
Whenever *Nmatrix* appears in the program, one has to multiply the subscript by −1. Since a negative subscript is not permitted, an additional statement must precede the subscripted variable.

example:

integer array *KURT*[−5:−1];
. . .
. . .
. . .
KURT[*K*]: = *N* ÷ *M*;

DIMENSION KURT(5)
. . .
. . .
. . .
KK = −*K*
KURT(*KK*) = *N*/*M*

---

integer array *matrix*[1:*Nc*]; *DIMENSION Nmatrix*(*Nc*)

R7
$Nc \leqslant \beta$

---

integer array *matrix*[0:*Nc*]; *DIMENSION Nmatrix*→
←[*Nc* + 1]

R7,N44
$Nc + 1 \leqslant \beta$

Whenever *Nmatrix* appears in the program the subscript must be increased by +1.

example:

integer array *ARL*[0:35];
. . .
. . .
. . .
ARL[*L*]: = *L*/*M* + *N*;

DIMENSION JARL(36)
. . .
. . .
. . .
JARL(*L* + 1) = *L*/*M* + *N*

---

integer array *matrix*[**if** $a_1 < a$ **then** Nc_1 **else** Nc_2:Nc_3];

```
      DIMENSION Nmatrix→
     ←(Nc3)
```

R7
$Nc_3 \leqslant \beta$

---

integer array *matrix*[Nc_1: **if** $a \geqslant a_1$ **then** Nc_2 **else** Nc_3];

```
      IF(a - a1)n1,n,n
n1|   DIMENSION Nmatrix→
     ←(Nc3)
      GO TO n2
 n|   DIMENSION Nmatrix→
     ←(Nc2)
n2|   CONTINUE
```

R7
$Nc_2,Nc_3 \leqslant \beta$

---

integer array *matrix1,matrix2, matrix3*[*1:Nc*];

```
      DIMENSION Nmatrix1→
     ←(Nc),Nmatrix2(Nc),→
     ←Nmatrix3(Nc)
```

R7
$Nc \leqslant \beta$

---

integer array *matrix*[Nc_1:Nc_2, Nc_3:Nc_4];

```
      DIMENSION Nmatrix→
     ←(Nc2,Nc4)
```

R7
$Nc_2,Nc_4 \leqslant \beta$

---

integer array *matrix*[*1:*Nc_1,*1:* Nc_2];

```
      DIMENSION Nmatrix→
     ←(Nc1,Nc2)
```

R7
$Nc_1,Nc_2 \leqslant \beta$

---

integer array *matrix*[Nc_1:Nc_2, Nc_3:Nc_4,Nc_5:Nc_6];	*DIMENSION Nmatrix*→ ←(Nc_2,Nc_4,Nc_6) R7 Nc_2,Nc_4,Nc_6 ⩽ β
integer array *matrix*[*1*:Nc_1,*1*:Nc_2,*1*:Nc_3];	*DIMENSION Nmatrix*→ ←(Nc_1,Nc_2Nc_3) R7 Nc_1,Nc_2,Nc_3 ⩽ β Restriction: IBM 1401 and IBM 1620_I permit only a maximum of two dimensions for an array.
integer array *matrix*[Nc_1:Nc_2, Nc_3:Nc_4, . . . , Nc_{k-1}:Nc_k];	For $k > 6$ no translation possible for IBM processors.
integer array *matrix1*,*matrix2*, *matrix3*[*1*:Nc_1],*matrix4*,*matrix5* [*1*:Nc_2,*1*:Nc_3];	*DIMENSION Nmatrix1*→ ←(Nc_1),*Nmatrix2*(Nc_1),→ ←*Nmatrix3*(Nc_1),*Nmatrix4*→ ←(Nc_2,Nc_3),*Nmatrix5*→ ←(Nc_2,Nc_3) R7 Nc_1,Nc_2,Nc_3 ⩽ β
integer procedure *token1*; see also: **integer procedure** *token*(. . , . . , *token1*, . . , . .);	This statement indicates that the parameter of the subprogram is the identifier of a subprogram. Omit in translation.

integer procedure *token;*	*SUBROUTINE token*
. . .	*COMMON* $v_i, \ldots, v_j$
. . .	. . .
. . .	. . .
end;	. . .
	RETURN
	END
	R5,R6,N21
	FIV:
	The same as above. R5,R6,N21, N29

integer procedure *token*(. . , . . ,	*FUNCTION Ntoken*→
$c_1, \ldots, c_k$, . . , . .);	←(. . , . . , $c_1, \ldots,$→
. . .	←c_k, . . , . .)
. . .	. . .
. . .	. . .
end;	. . .
	RETURN
	END
N22	
	R5,R6
	FIV:R5,N16,N29,N31

integer procedure *token*(. . , . . ,	*FUNCTION Ntoken*→
$v_1, \ldots, v_k$, . . , . .);	←(. . , . . , $v_1, \ldots,$→
. . .	←v_k, . . , . .)
. . .	*COMMON* $v_i, \ldots, v_j$
. . .	. . .
end;	. . .
	. . .
	RETURN
N22	*END*
	R5,R6,N21
	FIV:
	R5,N16,N21,N29,N31

integer procedure *token*(. . , . . ,
Nmatrix, . . , . .);
. . .
integer array *Nmatrix;*
. . .
. . .
. . .
end;

FUNCTION Ntoken→
←(. . , . . , *Nmatrix*, . . , . .)
COMMON $v_i, \ldots, v_j$
DIMENSION Nmatrix(*Nc*)
. . .
. . .
. . .
RETURN
END

N22

R5,R6,N21

Nc is the same number as in the *DIMENSION Nmatrix*(*Nc*) declaration in the main program. (Sometimes it is possible to use a smaller number for *Nc* in the subprogram.)
FIV:
R5,N16,N21,N29,N31

Nc can be replaced by an arbitrary integer variable identifier *Nv*.

---

integer procedure *token*(. . , . . ,
'text', . . , . .);
. . .
string *text;*
. . .
. . .
. . .
end;

FUNCTION Ntoken→
←(. . , . . , *cHtext*, . . , . .)
COMMON $v_i, \ldots, v_j$
. . .
. . .
. . .
RETURN
END

c is the number of fields occupied by *text*.

R5,R6,N21
FIV:
R5,N16,N21,N29,N31

integer procedure *token*(. . , . . ,
token1, . . , . .);
. . .
procedure *token1*;
. . .
. . .
. . .
end;

N22

Only some processors permit the use of a subprogram identifier as an argument. This requires special declarative information on "F" cards, see the appropriate instruction manual.

FIV:

INTEGER FUNCTION→
←token(. . , . . , *token1*,→
←. . , . .)
COMMON $v_i, \ldots, v_j$
. . .
. . .
. . .
RETURN
END

R5,N21,N29,N31
The call for the subprogram must be preceded by the declaration *EXTERNAL token1*

---

integer procedure *token*(v,v_1,c,)
text:(a_1,v_2,a_2);
. . .
. . .
. . .
end;

N22

FUNCTION Ntoken→
←(v,v_1,c,a_1,v_2,a_2)
COMMON $v_i, \ldots, v_j$
. . .
. . .
. . .
RETURN
END

R5,R6,N21
FIV:
R5,N16,N21,N29,N31

---

integer procedure *token*(. . , . . . , . .)
text:(. . , . . . , . .)*text1*:(. . , . . . , . .);
. . .
. . .
. . .
end;

N22

FUNCTION Ntoken→
←(. . , . . . , . . , . . . , . . . ,→
←. . , . . , . . . , . .)
COMMON v_i, . . . , v_j
. . .
. . .
. . .
RETURN
END

R5,R6,N21
FIV:
R5,N21,N29,N31

---

integer procedure *token*(v_1,v_2,a_1)
text:(c,a_2,v_3)*text1*:(c_1,c_2);

. . .
end;

N22

FUNCTION Ntoken→
←(v_1,v_2,a_1,c,a_2,v_3,c_1,c_2)
COMMON v_i, . . . , v_j
. . .
. . .
. . .
RETURN
END

R5,R6,N21
FIV:
R5,N16,N21,N29,N31

---

label *Nc*;
see also:
procedure *token*(. . , . . , *Nc*, . . , . .);

This statement indicates that the parameter *Nc* of the subprogram is a label.
Omit in translation.

---

label *v*;
see also:
procedure *token*(. . , . . , *v*, . . , . .);

This statement indicates that the parameter *v* of the subprogram is a label.
Omit in translation.

. . . *ln*(*a*) . . . ;	. . . *ALOG*(*Aa*) . . . R1 Exception: Some processors such as IBM 1401 and IBM 1620 require: . . . *LOGF*(*Aa*) . . .
. . . *ln*(*term*) . . . ;	. . . *ALOG*(*Aterm*) . . . R1 *Aterm* :N2 Exception: Some processors such as IBM 1401 and IBM 1620 require: . . . *LOGF*(*Aterm*) . . .
logical operators: ≡ ⊃ *V* Λ ¬	FIV: Not to be translated directly Not to be translated directly* *.OR.* *.AND.* *.NOT.* The logical expression to which *.NOT.* applies must be enclosed in parentheses if it contains two or more quantities.
logical values: **true** **false**	FIV: *.TRUE.* *.FALSE.*
multiplication: $a_1 \times a_2$	a_1*a_2 N2

* For example: Lterm1 ⊃ Lterm2 can be translated by: Lterm2 .OR.(.NOT.(Lterm1)).

numbers:

0	0
123	123
.1234	.1234
+0.1234	+0.1234
−123.456	−123.456
$+1.23_{10}5$	+1.23E5
$9.87_{10}+12$	9.87E+12
$2_{10}-3$	2.E−3
$-.012_{10}-03$	−0.012E−03
$-_{10}8$	−1.E8
$_{10}-5$	1.E−5
$+_{10}+5$	+1.E+5

Range for real numbers: ϵ
Range for integer numbers: δ
Range for exponents: η

---

procedure declaration:
see: **procedure** . . .; or **integer procedure** . . . ; or **real procedure** . . . ; or **Boolean procedure** . . . ;

---

procedure *token1;*
see also:
procedure *token*(. . , . . , *token1*, . . , . .);

This statement indicates that the parameter *token1* of the subprogram is the identifier of a subprogram.
Omit in translation.

---

procedure *token;*	*SUBROUTINE token*
. . .	*COMMON* $v_i, \ldots, v_j$
. . .	. . .
. . .	. . .
end;	. . .
	RETURN
	END

R5,R6,N21
FIV:
The same as above. R5,N21,N29

---

procedure *token*(. . , . . , c_1, . . . ,
c_k, . . , . .);
. . .
. . .
. . .
end;

SUBROUTINE token→
←(. . , . . , c_1, . . . , c_k, . . , . .)
. . .
. . .
. . .
RETURN
END

N22

R5,R6
FIV:
The same as above. R5,N29

---

procedure *token*(. . , . . , v_1, . . . ,
v_k, . . , . .)·
. . .
. . .
. . .
end;

SUBROUTINE token→
←(. . , . . , v_1, . . . , v_k, . . , . .)
COMMON v_i, . . . , v_j
. . .
. . .
. . .
RETURN
END

N22

R5,R6,N21
FIV:
The same as above. R5,N21,N29

---

procedure *token*(. . , . . , *v*,
. . , . .);
. . .
label *v*;
. . .
. . . .
go to *v*;

SUBROUTINE token→
←(. . , . . , *v*, . . , . .)
COMMON v_i, . . . , v_j
Nv = *v*
ASSIGN n TO Nv
. . .
. . .

. . .	*GO TO n*
. . .	*RETURN*
. . .	. . .
end;	. . .
	. . .
	RETURN
N22	*END*

R5,R6,N21

Exception: IBM 1401 and IBM 1620_I, in this case: the label *v* must be replaced by an integer number. This number must then agree with the appropriate number in the *GO TO* statement.
FIV:
The same as above. R5,N21,N29

---

procedure *token*(. . , . . , *Nc*, . . , . .);	*SUBROUTINE token*→
. . .	←(. . , . . , *Nc*, . . , . .)
label *Nc*;	*COMMON* $v_i, \ldots, v_j$
. . .	. . .
. . .	. . .
go to *Nc*;	*GO TO Nc*
. . .	*RETURN*
. . .	. . .
. . .	. . .
end;	. . .
	RETURN
	END
N22	

R5,R6,N21
FIV:
The same as above. R5,N21,N29

---

procedure *token*(. . , . . , *Nmatrix*,
. . , . .);
. . .
integer array *Nmatrix;*
. . .
. . .
. . .
end;

SUBROUTINE token→
←(. . , . . , *Nmatrix*, . . , . .)
COMMON $v_i, \ldots, v_j$
DIMENSION Nmatrix(*Nc*)
. . .
. . .
. . .
RETURN
END

N22

R5,R6,N21
Nc is the same number as in the *DIMENSION Nmatrix*(*Nc*) declaration in the main program. Sometimes it is possible to use a smaller number for *Nc* in the subprogram.
FIV:
The same as above. R5,N21,N29 *Nc* can be replaced by an arbitrary integer variable identifier *Nv*.

---

procedure *token*(. . , . . , 'text',
. . , . .);
. . .
string *text;*
. . .
. . .
. . .
end;

SUBROUTINE token→
←(. . , . . , *cHtext*, . . , . .)
COMMON $v_i, \ldots, v_j$
. . .
. . .
. . .
RETURN
END

c is the number of fields occupied by *text*.

R5,R6,N21
FIV:
The same as above. R5,R6,N21, N29

---

procedure *token*(. . , . . , *token1*,
. . , . .);
. . .
procedure *token1;*
. . .
. . .
. . .
end;

N22

Only some processors permit the use of subprogram identifiers as an argument. This requires special declarative information on "F" cards, see the appropriate instruction manual.

FIV:

SUBROUTINE token→
←(. . , . . , *token1*, . . , . .)
COMMON v_i, . . . , v_j
. . .
. . .
. . .
RETURN
END

R5,N21,N29

The call for the subprogram must be preceded by the declaration *EXTERNAL token1.*

---

procedure *token*(v,v_1,a_3)*text:*(a_1,v_2,
a_2);
. . .
. . .
. . .
end;

N22

SUBROUTINE token(v,→
←v_1,a_3,a_1,v_2,a_2)
COMMON v_i, . . . , v_j
. . .
. . .
. . .
RETURN
END

R5,R6,N21

FIV:

The same as above. R5,N21,N29

---

procedure *token*(v_1,v_2,a_1)*text1:*(c, a_2,v_3)*text2:*(c_1,c_2);
. . .
. . .
. . .
end;

SUBROUTINE token(v_1,→
←v_2,a_1,c,a_2,v_3,c_1,c_2)
COMMON V_i, . . . , v_j
. . .
. . .
. . .
RETURN
END

N22

R5,R6,N21
FIV:
The same as above. R5,N21,N29

---

real v_1,v_2, . . . , v_k;

No type declaration necessary. The variable identifiers must be changed into:
Av_1,Av_2, . . . , Av_k throughout the program.

R2
FIV:
REAL v_1,v_2, . . . , v_k

N30

---

real array *matrix;*
see:
array *matrix;*

---

real array *matrix*[. . .];
identical with:
array *matrix*[. . .];

---

real procedure *token1;*
see also:
real procedure *token*(. . , . . ,
token1, . . , . .);

This statement indicates that the parameter *token1* of the subprogram is the identifier of a subprogram.
Omit in translation.

---

real procedure *token;*
. . .
. . .
. . .
end;

SUBROUTINE token
COMMON $v_i, \ldots, v_j$
. . .
. . .
. . .
RETURN
END

R5,R6,N21
FIV:
The same as above. R5,N21,N29

---

real procedure *token*(. . , . . , c_1,
. . . , c_k, . . , . .);
. . .
. . .
. . .
end;

FUNCTION Atoken(. . ,→
←. . , c_1, . . . , c_k, . . , . .)
. . .
. . .
. . .
RETURN
END

N22

R5,R6
FORTRAN IV:

R5,N28,N29,N32

---

real procedure *token*(. . , . . , v_1, . . . , v_k, . . , . .);
. . .
. . .
. . .
end;

FUNCTION Atoken(. . ,→
←. . , v_1, . . . , v_k, . . , . .)
COMMON v_i, . . . , v_j
. . .
. . .
. . .
RETURN
END

N22

R5,R6,N21
FIV:

R5,N21,N28,N29,N32

---

real procedure *token*(. . , . . ,
Amatrix, . . , . .);
. . .
array *Amatrix*;
. . .
. . .
. . .
end;

FUNCTION Atoken(. . ,→
←. . , *Amatrix*, . . , . .)
COMMON v_i, . . . , v_j
DIMENSION Amatrix(*Nc*)
. . .
. . .
. . .
RETURN
END

N22

R5,R6,N21
Nc is the same number as in the *DIMENSION Amatrix*(*Nc*) declaration in the main program. Sometimes it is possible to use a smaller number for *Nc* in the subprogram.
FIV:

R5,N21,N28,N29,N32

Nc can be replaced by an arbitrary integer variable identifier *Nv*.

real procedure *token*(. . , . . ,	*FUNCTION Atoken*(. . ,→
'text', . . , . .);	←. . , *cHtext*, . . , . .)
. . .	*COMMON* v_i, . . . , v_j
string *text;*	. . .
. . .	. . .
. . .	. . .
. . .	*RETURN*
end;	*END*
	c is the number of fields occupied by *text*.
	R5,R6,N21
	FIV:
	R5,N21,N29,N32N28,

real procedure *token*(. . , . . ,	Only some processors permit the
token1, . . , . .);	use of a subprogram identifier as
. . .	an argument. This requires special
procedure *token1;*	declarative information on "F"
. . .	cards, see the appropriate instruc-
. . .	tion manual.
. . .	FIV:
end;	*REAL FUNCTION*→
	←*token*(. . , . . , *token1*, . . ,→
	←. .)
N22	*COMMON* v_i, . . . , v_j
	. . .
	. . .
	. . .
	RETURN
	END
	R5,N21,N29,N32
	The call for the subprogram must be preceded by the declaration *EXTERNAL token1*

real procedure *token*(v,v_1,c)*text:* (a_1,v_2,a_2);	*FUNCTION Atoken*(v,v_1,→ ←c,a_1,v_2,a_2)
. . .	*COMMON* v_i, . . . , v_j
. . .	. . .
. . .	. . .
end;	. . .
	RETURN
	END
N22	
	R5,R6,N21
	FIV:
	R5,N21,N28,N29,N32

real procedure *token*(v_1,v_2,a_1) *text:*(c,a_2,v_3)*text:*(c_1,c_2);	*FUNCTION Atoken*(v_1,→ ←v_2,a_1,c,a_2,v_3,c_1,c_2)
. . .	*COMMON* v_i, . . . , v_j
. . .	. . .
. . .	. . .
end;	. . .
	RETURN
	END
N22	
	R5,R6,N21
	FIV:
	R5,N21,N28,N29,N32

relational operators:	The translation of the relational operators can be found in connection with the if statements.
	FIV:
$<$	*.LT.*
$\leqslant$	*.LE.*
$=$	*.EQ.*
$\geqslant$	*.GE.*
$>$	*.GT.*
$\neq$	*.NE.*

. . . *sign*(*a*) . . . ;	*IF*(*a*)n_1,n_2,n_3
	n_1| *Na* = −1
	GO TO n
	n_2| *Na* = 0
	GO TO n
	n_3| *Na* = 1
	n| *CONTINUE*
	. . . *Na* . . .

	FIV:
. . . *sign*(*Aa*) . . . :	. . . *SIGN*(*Aa*) . . .
. . . *sign*(*Nc*) . . . :	. . . *ISIGN*(*Na*) . . .
. . . *sign*(*term*) . . . :	*IF*(*term*)n_1,n_2,n_3
	n_1| *Na* = −1
	GO TO n
	n_2| *Na* = 0
	GO TO n
	n_3| *Na* = 1
	n| *CONTINUE*
	. . . *Na* . . .

. . . *sin*(*a*) . . . ;	. . . *SIN*(*Aa*) . . .
	R1,N42

. . . *sin*(*term*) . . . ;	. . . *SIN*(*Aterm*) . . .
	R1,N42
	Aterm: N2

. . . *sqrt(a)* . . . ;	. . . *SQRT(Aa)* . . .
	R1,N42
. . . *sqrt(term)* . . . ;	. . . *SQRT(Aterm)* . . .
	R1,N42 *Aterm:* N2
statement label: *n:* . . . ;	*n*| . . . $n \leqslant \zeta$ Only columns 1 through 5 can contain a statement label. Exception: IBM 1620, in this case only columns 2 through 5 can contain a statement label.
statement label: *a:* . . . ;	*n*| . . . $n \leqslant \zeta$ If *a* is an alphabetic character, it must be replaced by an integer number, this number must then agree with the appropriate number in the *GO TO* statement. Only columns 1 through 5 can contain a statement label. Exception: IBM 1620, in this case only columns 2 through 5 can contain a statement label.
statement separator: ;	No separator, different statements must be on different records.

string *text;* see also: **procedure** *token*(. . , . . , *'text'*, . . , . .);	This statement indicates that *text* is used as a parameter of the subprogram. Omit in translation.

subtraction:

$a_1 - a_2$	$a_1 - a_2$
	N2

switch declaration:
see: **switch** *jump:* = . . , . . . , . . ;

switch *jump;*	This statement indicates that the parameter *jump* of the subprogram is the identifier of a switch. Omit in translation.

switch *jump:* = $Nc_1, \ldots, Nc_k$;	
. . .	. . .
. . .	. . .
. . .	. . .
go to *jump*[Nv];	*GO TO*($Nc_1, \ldots, Nc_k$),Nv
. . .	. . .
. . .	. . .
. . .	. . .
Nc_1: . . .	Nc_1| . . .
Nc_2: . . .	Nc_2| . . .
.	.
.	.
.	.
Nc_k: . . .	Nc_k| . . .
. . .	. . .
. . .	. . .
. . .	. . .
	$Nv \leqslant 10$ No switch declaration necessary.

switch *jump*: = $v_1, \ldots, v_k$;

```
. . .
. . .
. . .
go to jump[Nv];
. . .
. . .
. . .
v1: . . .
v2: . . .
.
.
.
vk: . . .
. . .
. . .
. . .
```

```
. . .
. . .
. . .
GO TO(n1, . . . , nk),Nv
. . .
. . .
. . .
n1| . . .
n2| . . .
.
.
nk| . . .
. . .
. . .
. . .
```

$Nv \leqslant 10$

No switch declaration necessary. The statement labels $v_1, \ldots, v_k$ must be replaced by the statement labels $n_1, \ldots, n_k$ throughout the program.

---

switch *jump*: = v_1,v_2,Nc, **if** $a \neq a_1$ **then** Nc_1 **else** Nc_2;

```
. . .
. . .
. . .
go to jump[Nv];
. . .
. . .
v1: . . .
v2: . . .
Nc: . . .
Nc1: . . .
Nc2: . . .
```

```
. . .
. . .
. . .
GO TO(n1,n2,Nc,n3),Nv
n3| IF(a − a1)Nc1,Nc2,Nc1
. . .
. . .
n1| . . .
n2| . . .
Nc| . . .
Nc1| . . .
Nc2| . . .
```

$Nv \leqslant 10$

No switch declaration necessary. The statement labels v_1,v_2 must be replaced by the statement labels n_1,n_2 throughout the program.

switch *jump*: = *v*,*Nc*,v_1,v_2,*matrix*
[Nv_2],Nc_2;
. . .
. . .
. . .
go to *jump*[*Nv*];
. . .
. . .
v: . . .
Nc: . . .
v_1: . . .
v_2: . . .
Nc_i: . . .
.
.
.
Nc_k: . . .
Nc_2: . . .

Nc_i, . . . , Nc_k are the elements of *matrix*[. . .].

. . .
. . .
. . .
GO TO(*n*,*Nc*,n_1,n_2,n_7,→
←Nc_2),*Nv*
. . .
. . .

n	. . .
Nc	. . .
n_1	. . .
n_2	. . .
n_7	Nv_4 = *Nmatrix*(Nv_2)
	GO TO(Nc_i, . . . , Nc_k),Nv_4
Nc_i	. . .
.	
.	
.	
Nc_k	. . .
Nc_2	. . .

Nv,Nv_4 ⩽ 10

No switch declaration necessary. The statement labels *v*,v_1,v_2 must be replaced by the statement labels *n*,n_1,n_2 throughout the program.

. . . *token*(. . , . . . , . .) . . . ;
token is the identifier of a previously defined **Boolean procedure**.

No logical statements possible in FORTRAN II.
FIV:

LOGICAL token
. . .
COMMON v_i, . . . , v_j
. . . *token*(. . , . . . , . .) . . .

N21

. . . *token*(. . , . . . , . .) . . . ;
token is the identifier of a previously defined **integer procedure**.

COMMON v_i, . . . , v_j
. . . *Ntoken*(. . , . . . , . .) . . .

N21
FIV:
INTEGER token
. . .
COMMON v_i, . . . , v_j
. . . *token*(. . , . . . , . .) . . .

N21

. . . *token*(. . , . . . , . .) . . . ;
token is the identifier of a previously defined **real procedure**.

COMMON v_i, . . . , v_j
. . . *Atoken*(. . , . . . , . .) . . .

N21
FIV:
REAL token
. . .
COMMON v_i, . . . , v_j
. . . *token*(. . , . . . , . .) . . .

N21

. . . *token*(. . , . . , *token1*, . . , . .)
. . . ;
token is the identifier of a previously defined **real procedure** or **integer procedure** or **Boolean procedure**.

FIV:
REAL token,token1
EXTERNAL token1
. . .
. . . *token*(. . , . . , *token1*,→
←. . , . .) . . .

N21

The type declaration *REAL token, token1* might be changed in agreement with the type of the subprograms *token* and *token1*.

token(. . , . . . , . .);	*COMMON* v_i, . . . , v_j
token is the identifier of a previously defined **procedure**.	*CALL token*(. . , . . . , . .)
N22	N21

token(. . , . . . , *token1*, . . , . .);	FIV:
token is the identifier of a previously defined **procedure**.	*EXTERNAL token1* *COMMON* v_i, . . . , v_j *CALL token*(. . , . . . ,→ ←*token1*, . . , . .)

token(. . , . . . , . .)*text:*(. . , . . . , . .);	*COMMON* v_i, . . . , v_j *CALL token*(. . , . . . , . . ,→ ←. . , . . . , . .)
token is the identifier of a previously defined **procedure**.	
N22	N21

token(. . , . . . , . .)*text1:*(. . , . . . , . .)*text2:*(. . , . . . , . .);	*COMMON* v_i, . . . v_j *CALL token*(. . , . . . , . . ,→ ←. . , . . . , . . , . . , . . . , . .)
token is the identifier of a previously defined **procedure**.	
N22	N21

. . . **true** . . . ;	FIV: . . . .*TRUE*. . .

type declaration:
see: **integer** v_1, . . . , v_k;
or **real** v_1, . . . , v_k;
or **Boolean** v_1, . . . , v_k;

value v_1, . . . , v_k;	Omit in translation.

(integer) variable identifier:
The variable identifier must be listed in an integer declaration.

The first character of the variable identifier must be *I,J,K,L,M*, or *N*.
The maximum length of the variable identifier is restricted to α characters.

N10
examples:

K	*K*
B	*NB*
G42	*NG42*
E803	*NE803*
IBM	*IBM*

N3
FIV:
The variable identifier must be listed in an *INTEGER* declaration. If it is declared under *INTEGER*, then the variable identifier can begin with an arbitrary alphabetic character.
N10,N30

(real) variable identifier:
The variable identifier must be listed in a real declaration.

The first character of the variable identifier cannot be *I,J,K,L,M*, or *N*.
The maximum length of the variable identifier is restricted to α characters.

N10

	examples:	
	E605	*E605*
	LANDAU	*ANDAU*
	Z	*Z*
	M	*AM*
	A12F	*A12F*
	N4 FIV: The variable identifier must be listed in a *REAL* declaration. If it is declared under *REAL*, then the variable identifier can begin with an arbitrary alphabetic character.	
	N10,N30	
(subscripted) variable identifier:	The same, only the subscript appears in round brackets. The maximum value for a subscript is β.	
	N10	
	examples:	
$X[1]$		$X(1)$
$T[Z]$		$T(NZ)$
$HANS[38]$		$HANS(38)$
$BETA[I,N]$		$BETA(I,N)$
$KALLA[8,M,105]$		$KALLA(8,M,105)$
$M30[I]$		$M30(I)$
$A1[R + 1,L]$		$A1(NR + 1,L)$
$DELTA[3 \times M - 5]$		$DELTA(3*M - 5)$

Part 5

FORTRAN II and IV → ALGOL 60

On the following pages the left column contains FORTRAN II and FORTRAN IV statements (the latter are indicated by a preceding FIV:) and the right column the appropriate ALGOL 60 statements. The sequence of statements is in alphabetical order, except where this rule is broken to place together statements which belong together. Some parts of a computer program (such as subprograms or *GO TO* (. . .), *Nv* or *ASSIGN*, etc.) are easier to translate when treated as a group of statements; such statements are grouped together. Groups with essential combinations of statements inside this group are translated first. After this, a few examples show how these groups can be combined together. The meaning of the notes (abbreviated by N) and restrictions (abbreviated by R) is given in Part 2.2 and 2.3. Statements which cannot be translated directly, since they refer to special machine features (such as *SENSE SWITCH*, *BACKSPACE*, etc.), are explained. Since ALGOL 60 does not include input/output statements, the examples are given with the input/output statements of the National Elliott 803 (NE 803).

Important remark: Occasionally, the length of a FORTRAN statement exceeds the width of a line. To indicate that the next line should appear as a continuation of the previous line, we use two arrows → ←.

The following example makes this clear:
The statement

DIMENSION Amatrix(Nc_1),→
←*Amatrix*(Nc_2)

should actually be written as:

DIMENSION Amatrix(Nc_1),*Amatrix*(Nc_2)

FIV,FII:	
. . . *ABS*(*a*) . . .	. . . *abs*(*a*) . . . ;
a:N12	N1
. . . *ABSF*(*a*) . . .	. . . *abs*(*a*) . . . ;
a:N12,N42	N1
ACCEPT n,a	This statement is used to allow *a* to be read in from the console typewriter in accordance with the format statement *n*.
ACCEPT TAPE n,a	This statement causes *a* to be read from the paper tape reader, in accordance with the format statement *n*.
addition:	
$a_1 + a_2$	$a_1 + a_2$
FIV:	
. . . *AINT*(*Aa*) . . .	. . . *entier*(*Aa*) . . . ;
FIV,FII:	
. . . *ALOG*(*a*) . . .	. . . *ln*(*a*) . . . ;
a:N12	
FIV:	
. . . *ALOG10*(*a*) . . .	. . . *0.4342945* × *ln*(*a*) . . . ;
a:N12	

FIV:

... *.AND.* ...	... Λ ... ;

arithmetic operators:

+	+
−	−
*	×
/	/
**	↑

assignment statement:

$a_1 = a_2$	$a_1 := a_2$;

a_2:N12

FIV:
logical assignment statement:

$a.EQ.a_1$	$a = a_1$;

ASSIGN n TO Nv	Omit in translation.

FIV, FII:

... *ATAN*(*a*) ...	... *arctan*(*a*) ... ;

a:N12

... *ATANF*(*a*) ...	... *arctan*(*a*) ... ;

a:N12

FIV:

BACKSPACE Na	This statement causes the magnetic tape unit *Na* to backspace one record.

C (in column 1) *text*	; **comment** *text* ; For the computation, this is equivalent to: ; Restriction: *text* may not include a semicolon. Alternative translation: directly after an **end** statement. **end** *text* For the computation, this is equivalent to: **end** Restriction: *text* may not include **end** or ; or **else**. Alternative translation: **begin comment** *text* ; For the computation, this is equivalent to: **begin** Restriction: *text* may not include a semicolon.
FIV:	
CALL DUMP($v_1, \ldots, v_k$)	This causes the indicated limits of the core storage to be dumped and execution to be terminated, but allows the computer to continue on its next program.
CALL EXIT	Terminates execution, but allows the computer to continue on its next program.
CALL LINK token	This statement is used to call the subprogram *token* from magnetic disk storage and transfer is made to the first executable statement.

FIV:

CALL PDUMP($v_1, \ldots, \rightarrow$ $\leftarrow v_k$)

This causes the indicated limits of core storage to be dumped and execution to be continued.

FIV:

CALL SSWTCH($Nv_1, \rightarrow$ $\leftarrow Nv_2$)

equivalent to:

IF(*SENSE SWITCH* $\rightarrow$ $\leftarrow Nc$)n_1, n_2

CALL token

token is the identifier of a *SUBROUTINE* without parameters.

token;

The declaration of the appropriate procedure must precede this statement.

CALL token(. . , . . . , . .)

token is the identifier of a *SUBROUTINE.*

token(. . , . . . , . .);

The declaration of the appropriate procedure must precede this statement.

FIV:

CALL token(. . , . . ,→ ←*token1*, . . , . .)

token is the identifier of a *SUBROUTINE.*

token(. . , . . , *token1*, . . , . .);

The declaration of the appropriate procedure must precede this statement.

COMMON $v_1, \ldots, v_k, \rightarrow$ ←*matrix*

or

COMMON $v_1/\ldots/v_k/\rightarrow$ ←*matrix*

Variables and/or array identifiers appearing in this statement can be shared by a main program and its subprograms.
Omit in translation.

FIV:

COMPLEX $v_1, \ldots, v_k, \rightarrow$ ←*token*

The declared variables and function are complex.
For writing complex numbers see: numbers.

CONTINUE	This is a dummy statement which results in no instructions to the program. It is most frequently used as the last statement in the range of a loop to provide a transfer address for *IF* and *GO TO* statements.

FIV, FII:

. . . *COS*(*a*) . . .	. . . *cos*(*a*) . . . ;

a:N12

. . . *COSF*(*a*) . . .	. . . *cos*(*a*) . . . ;

a:N12,N42

FIV:

DATA $v_1, \ldots, v_k(c_1,\rightarrow$	$v_1 := c_1;$
$\leftarrow \ldots, c_k)$	.
	.
	.
	$v_k := c_k;$
	example 1:
DATA A,B,NON,AR3,→	*A*: = −3.25;
←*Z*(−3.25,9.23E5,13,→	*B*: = $9.23_{10}5$;
←2.5,−1.2)	*NON*: = 13;
	AR3: = 2.5;
	Z: = −1.2;
	example 2:
DATA(*C*(*I*),*I* = 1,4)→	*C*[1]: = 2.3;
←(2.3,3.4,0.5,7.203)	*C*[2]: = 3.4;
	C[3]: = 0.5;
	C[4]: = 7.203;

FIV:

DATA $v_1,v_2,v_3(c_1,c_2,c_3),\rightarrow$	$v_1 := c_1;\ v_2 := c_2;\ v_3 := c_3;$
$\leftarrow v_4, \ldots, v_k(c_4, \ldots, c_k)$	$v_4 := c_4; \ldots; v_k := c_k;$

FIV:

DATA $v_1,v_2, \ldots, v_k(c_1,\rightarrow$	$v_1 := c_1; v_2 := c_2;$
$\leftarrow c_2/Nc/c_3,c_{Nc+3},c_{Nc+4},\rightarrow$	$v_3 := v_4 := \ldots := c_{Nc+2} := c_3;$
$\leftarrow \ldots, c_k)$	$v_{Nc+3} := c_{Nc+3}; \ldots; v_k := c_k;$
	example:
DATA B1,B2,B3,B4,→	*B1: = 5; B2: = 7.01;*
←*B5,T,S,U,V,W,Z(5,*→	*B3: = B4: = B5: = T: =*
←*7.01/7/3,2.5,3.735)*	*S: = U: = V: = 3;*
	W: = 2.5; Z: = 3.735;

FIV:

... *DATAN*(*a*) ...	... *arctan*(*a*) ... ;

N41
a:N12

FIV:

... *DCOS*(*a*) ...	... *cos*(*a*) ... ;

N41
a:N12

DEFINE DISK ...	This statement specifies to the processor the size and quantity of data records that will be used with a particular program.

FIV:

... *DEXP*(*a*) ...	... *exp*(*a*) ... ;

N41
a:N12

DIMENSION Amatrix→	**array** *Amatrix*[*1:Nc*];
←(*Nc*)	
	$Nc \leqslant \beta$

N26

	DIMENSION Nmatrix→ *←(Nc)*	**integer array** *Nmatrix[1:Nc];* *Nc* ⩽ β
N26		
	DIMENSION Amatrix→ *←(Nc₁,Nc₂)*	**array** *Amatrix[1:Nc₁,1:Nc₂];* *Nc₁,Nc₂* ⩽ β
N26		
	DIMENSION Nmatrix→' *←(Nc₁,Nc₂)*	**integer array** *Nmatrix[1:Nc₁,* *1:Nc₂];*
N26		*Nc₁,Nc₂* ⩽ β
	DIMENSION Amatrix→ *←(Nc₁,Nc₂,Nc₃)*	**array** *Amatrix[1:Nc₁,1:Nc₂,* *1:Nc₃];* *Nc₁,Nc₂,Nc₃* ⩽ β
N26		
	DIMENSION Nmatrix→ *←(Nc₁,Nc₂,Nc₃)*	**integer array** *Nmatrix[1:Nc₁,* *1:Nc₂,1:Nc₃];* *Nc₁,Nc₂,Nc₃* ⩽ β
N26		

DIMENSION Amatrix1→ ←(*Nc*),*Amatrix2*(*Nc*),→ ←*Amatrix3*(*Nc*)	**array** *Amatrix1,Amatrix2,* *Amatrix3*[*1:Nc*];
	$Nc \leqslant \beta$
N26	

DIMENSION Nmatrix1→ ←(Nc_1),*Nmatrix2*(Nc_2),→ ←*Nmatrix3*(Nc_3)	**integer array** *Nmatrix1*[*1:*Nc_1], *Nmatrix2*[*1:*Nc_2],*Nmatrix3* [*1:*Nc_3];
	$Nc_1,Nc_2,Nc_3 \leqslant \beta$
N26	

DIMENSION Amatrix1→ ←(Nc_1),*Nmatrix*(Nc_2,→ ←Nc_3),*Amatrix2*(Nc_4,→ ←Nc_5,Nc_6)	**array** *Amatrix1*[*1:*Nc_1], *Amatrix2*[*1:*Nc_4,*1:*Nc_5,*1:*Nc_6]; **integer array** *Nmatrix*[*1:*Nc_2, *1:*Nc_3];
	$Nc_1, \ldots, Nc_6 \leqslant \beta$

division: a_1/a	a_1/a
a,a_1:N12 exception: Na_1/Na	$Na_1 \div Na$ if the arithmetic operator $\div$ is not available then translate: *sign*(Na_1/Na) × *entier*(*abs* (Na_1/Na))

FIV:

... *DLOG*(*a*) ...	... *ln*(*a*) ... ;

N41
a:N12

FIV:

... *DLOG10*(*a*) ...	... *0.4342945* × *ln*(*a*) ... ;

N41
a:N12

DO n Nv = m_1,m_3	**for** *Nv*: = m_1 **step** *1* **until** m_3 **do**
...	**begin**
...	...
...	...
	...
n| ...	
	...
	end ;
	N18

DO n Nv = m_1,m_3,m_2	**for** *Nv*: = m_1 **step** m_2 **until** m_3 **do**
...	**begin**
...	...
...	...
	...
n| ...	
	...
	end ;
	N18

FIV:

DOUBLE PRECISION→ ←$v_1, \ldots, v_k$, token	The declared variables and functions are calculated with a precision equivalent to twice as many significant digits as are obtained in ordinary operation.

FIV:

. . . *DSIN*(*a*) . . .	. . . *sin*(*a*) . . . ;

N41
a:N12

FIV:

. . . *DSQRT*(*a*) . . .	. . . *sqrt*(*a*) . . . ;

N41
a:N12

END	**end**

FIV:

END FILE Na	This statement causes an end-of-file mark to be written on the tape of the magnetic tape unit *Na*.

FIV:

. . . .*EQ*. . . .	. . . = . . . ;
EQUIVALENCE(v_1,v_2)	$v_1 := v_2$;
EQUIVALENCE(v_1,v_2,v_3)	$v_1 := v_2 := v_3$; v_1,v_2,v_3:N2 If N2 is violated then translate: $v_1 := v_3$; $v_2 := v_3$;

EQUIVALENCE(v_1,v_2),→ ←(v_3,v_4,v_5)	v_1: = v_2; v_3: = v_4: = v_5; v_3,v_4,v_5:N2 If N2 is violated then translate: v_1: = v_2; v_3: = v_5; v_4: = v_5;

FIV, FII:

. . . *EXP*(a) . . .	. . . *exp*(a) . . . ;

a:N12

. . . *EXPF*(a) . . .	. . . *exp*(a) . . . ;

a:N12,N42

---

exponentiation:

a_1**a	a_1↑a

a,a_1:N12

---

expression:

An expression is of integer type if all the operands are integer. An expression is of real type if all the operands are real.	An expression is of integer type if all the operands are integer, otherwise the expression is of real type.

FIV:

EXTERNAL token1,→ ←*token2*,*token3*	This statement indicates, that the subprogram identifiers *token1*, *token2*,*token3* can be used as arguments in subprograms. Omit in translation.

FIV:

. . . .*FALSE*. . . .	. . . **false** . . . ;
FETCH(*c*) $a_1, \ldots, a_k$	This statement is used to read from the disc storage the data $a_1, \ldots, a_k$ from the record specified by *c*.
FIND(*a*)	This statement is used to position the access arm of a disk storage drive over the cylinder which contains the record *a*.
. . . *FLOAT*(*Na*) . . .	. . . *Na* . . . ;

FIV:

FORMAT(*Dw.d*)

The number in connection with this input/output statement appears as a double precision number with exponent on a field of width *w* with *d* digits to the right of the decimal point.

example:

```
n| FORMAT(D17.10)
   PUNCH n,C
```

7.1991567432D + 10

FORMAT(Ew.d)

This number in connection with this input/output statement appears as a real number with exponent on a field of width *w* with *d* digits to the right of the decimal point.

NE 803:

No translation of a format statement if this is connected with an input statement.

SCALED(w − 6)

example:

n|	*FORMAT(E10.3)* *PUNCH n,C* 2.997E + 10	*PRINT SCALED(4),C;* 2.997 @ + 10

FORMAT(Fw.d)

The number in connection with this input/output statement appears as a real number without exponent on a field of width *w* with *d* digits to the right of the decimal point.

NE 803:

No translation of a format statement if this is connected with an input statement.

ALIGNED(w − d − 2,d)

example:

```
n| FORMAT(F9.4)
   PUNCH n,C
   -938.2129
```

```
PRINT ALIGNED(3,4),C;
-938.2129
```

FORMAT(Iw)

The number in connection with this input/output statement appears as an integer number in a field of width *w*.
NE 803:

No translation of a format statement if this is connected with an input statement.

DIGITS(w − 1)

example:

```
n| FORMAT(I5)
   PUNCH n,K
    137
```

```
PRINT DIGITS(4),K;
 137
```

FIV:

FORMAT(Lw)

This statement in connection with an input statement causes a value of *.TRUE.* or *.FALSE.* to be assigned to the corresponding logical variable, depending whether the first character in the field is a *T* or an *F*.
Note: If the field of width *w* consists entirely of blanks, a value of *.FALSE.* is assumed.
This statement in connection with an output statement causes a *T* or an *F* inserted in the output record for the corresponding logical variable with a value of *.TRUE.* or *.FALSE.*. The single character is preceded by *w* − 1 blank fields.

example:

```
n| FORMAT(L6)
   PUNCH n,C
|     T|
```

if *C* has the value *.TRUE.*

or

```
|     F|
```

if *C* has the value *.FALSE.*

FORMAT(*c*X,{ },*e*X,→
←{ })

The two numbers in connection with this input/output statement appear on one record, the first number is preceded by *c* blank fields and the second number is preceded by *e* blank fields.

NE 803:

No translation of a format statement if it is connected with an input statement.

$*Sc?*,{ },*SAMELINE*,$v_1$,$*Se?*,
{ },v_2;

example:

n|	*FORMAT*(*5X,E10.3,*→ ←*12X,F5.2*) *PUNCH n,A,B*	*PRINT* $*S5?*,*SCALED*(*4*), *SAMELINE,A,*$*S12?*, *ALIGNED*(*1,2*),*B;*

FORMAT(*2*(*F10.5,*→
←*E10.3*),*I8*)

this is equivalent to:

FORMAT(*F10.5,E10.3,*→
←*F10.5,E10.3,I8*)

FORMAT({ }, . . . , { })

NE 803:

example:

n|	*FORMAT*(*F7.3,10X,*→ ←*F6.3,10X,E13.4*) *PUNCH n,A,B,C*	*PRINT ALIGNED*(*2,3*), *SAMELINE,A,*$*S10?*,*ALIGNED* (*1,3*),*B,*$*S10?*,*SCALED*(*7*),*C;*

	FORMAT(*cEw.d,eFw.d*)	The $c + e$ numbers in connection with this input/output statement appear on one record. The format of the first group of numbers is specified as *Ew.d* and the second group of numbers as *Fw.d*. NE 803: No translation of a format statement if it is connected with an input statement.

example:

n|	*FORMAT*(*3E10.3,2F5.2*) *PUNCH n,A,B,C,D,E*	*PRINT SCALED*(*4*), *SAMELINE,A,B,C,ALIGNED* (*1,2*),*D,E;*
	FORMAT(*cEw.d*/(*eFw.d*))	The format of the starting *c* numbers in connection with this input/output statement is specified as *Ew.d*, the remaining numbers appear on new records in groups of *e* numbers on one record and in the format *Fw.d*.
	FORMAT({ }///{ })	The first and second number in connection with this input/output statement are separated by 3 empty records. NE 803: No translation of a format statement if it is connected with an input statement.

example:

n|	*FORMAT(E10.3///F5.2)* *PUNCH n,A,B*	*PRINT SCALED(4),A,$L3?,* *ALIGNED(1,2),B;*
	3.321E + 33 ——— ——— 1.23	3.321 @ + 33 ——— ——— 1.23

FORMAT(cHtext) — *text* in connection with this input/output statement appears on a field of width *c*.

NE 803:

example:

n|	*FORMAT(14H THIS→* *←IS A TRIAL)* *PUNCH n* THIS IS A TRIAL	*PRINT $ THIS IS A TRIAL ?;* THIS IS A TRIAL

FORMAT({ },cHtext) — The number in connection with this input/output statement appears as specified under { } followed by *text* occupying a field of width *c*.

NE 803:

example:

```
n| FORMAT(I3,14H=→          PRINT DIGITS(2),K,$ =
 ←ATOMIC NUMBER)          ATOMIC NUMBER?;
 PUNCH n,K                75 = ATOMIC NUMBER
 75 = ATOMIC NUMBER
```

FORMAT(*c*H*text*{ }) — *text* in connection with this input/output statement appears on a field of width *c* followed by a number specified by{ }.

example:

NE 803:

```
n| FORMAT(20H→              PRINT $ SOMMERFELDS
 ←SOMMERFELDS→            CONSTANT = ?,SAMELINE,
 ← CONSTANT = F8.3)       ALIGNED(3,3),S;
 PUNCH n,S                SOMMERFELDS CONSTANT = 137.040
 SOMMERFELDS CONSTANT = 137.040
```

FORMAT(*Ac*,{ }, . . . ,→ ←{ }) — *c* alphameric characters appear followed by numbers specified by { }, . . . , { }.

example:

NE 803:

```
n| FORMAT(A7,3X,I3,→       PRINT $ COBALT?,
 ←5X,F6.2)                SAMELINE,$S3?,DIGITS(2),
 PUNCH n, COBALT,K,A      K,$S5?,ALIGNED(2,2),A;
```

FORMAT(1H +,{ }) or *FORMAT*(1H,{ })	This format statement indicates that the records appearing on the line-printer are single spaced.
FORMAT(1HO,{ })	This format statement indicates that the records appearing on the line-printer are double spaced.
FORMAT(1H1,{ })	This format statement indicates that the records appear on the next page of the line-printer.

FUNCTION Atoken→
←(. . , . . , c_1, . . . , c_k,→
←. . , . .)
COMMON . . , . . . , . .
. . .
. . .
. . .
RETURN
END

N27

real procedure *Atoken* (. . , . . ,
c_1, . . . , c_k, . . , . .);
. . .
begin
. . .
. . .
. . .
end;

N23

FUNCTION Ntoken→
←(. . , . . , c_1, . . . , c_k,→
←. . , . .)
COMMON . . , . . . , . .
. . .
. . .
. . .
RETURN
END

N27

integer procedure *Ntoken*(. . , . . ,
c_1, . . . , c_k, . . , . .);
. . .
begin
. . .
. . .
. . .
end;

N23

FUNCTION Atoken→	**real procedure** *Atoken* (. . , . . ,
←(. . , . . , v_1, . . . , v_k, . . ,→	v_1, . . . , v_k, . . , . .);
←. .)	**value** . . , v_1, . . . , v_k, . . ;
COMMON . . , . . . , . .	**real** . . , . . . , . . ;
. . .	**integer** . . , . . . , . . ;
. . .	. . .
. . .	**begin**
RETURN	. . .
END	. . .
	. . .
	end;
N27	N23,N24,N25

FUNCTION Ntoken→	**integer procedure** *Ntoken* (. . ,
←(. . , . . , v_1, . . . , v_k,→	. . , v_1, . . . , v_k, . . , . .);
←. . , . .)	**value** . . , v_1, . . . , v_k, . . ;
COMMON . . , . . . , . .	**real** . . , . . . , . . ;
. . .	**integer** . . , . . . , . . ;
. . .	. . .
. . .	**begin**
RETURN	. . .
END	. . .
	. . .
	end;
N27	N23,N24,N25

FUNCTION Atoken→	**real procedure** *Atoken* (. . , . . ,
←(. . , . . , *Nmatrix*, . . ,→	*Nmatrix*, . . , . .);
←. .)	. . .
COMMON . . , . . . , . .	**integer array** *Nmatrix*;
DIMENSION Nmatrix→	. . .
←(*Nc*)	**begin**
. . .	. . .
. . .	. . .
. . .	. . .
RETURN	**end**;
END	
N27	N23

```
FUNCTION Ntoken→                integer procedure Ntoken (. . , . . ,
←(. . , . . , Nmatrix, . . ,→   Nmatrix, . . , . .);
←. .)                           . . .
COMMON . . , . . . , . .        integer array Nmatrix;
DIMENSION Nmatrix→              . . .
←(Nc)                           begin
. . .                           . . .
. . .                           . . .
. . .                           . . .
RETURN                          end;
END
```

N27 N23

```
FUNCTION Atoken→                real procedure Atoken(. . , . . ,
←(. . , . . , cHtext, . . ,→    'text', . . , . .);
←. .)                           . . .
COMMON . . , . . . , . .        string text;
. . .                           . . .
. . .                           begin
. . .                           . . .
RETURN                          . . .
END                             . . .
                                end;
```

N27 N23

```
FUNCTION Ntoken→                integer procedure Ntoken(. . , . . ,
←(. . , . . , cHtext, . . ,→    'text', . . , . .);
←. .)                           . . .
COMMON . . , . . . , . .        string text;
. . .                           . . .
. . .                           begin
. . .                           . . .
RETURN                          . . .
END                             . . .
                                end;
```

N27 N23

FUNCTION Atoken→	**real procedure** *Atoken* (Av_1,Av_2,
←(Av_1,Av_2,Nv_3,c_1,c_2,→	Nv_3,c_1,c_2,*Amatrix*);
←*Amatrix*)	**value** Av_1,Av_2,Nv_3;
COMMON . . , . . . , . .	**real** Av_1,Av_2;
DIMENSION Amatrix→	**integer** Nv_3;
←(*Nc*)	**array** *Amatrix*;
. . .	**begin**
. . .	. . .
. . .	. . .
RETURN	. . .
END	**end**;

N27 N23,N24,N25

FUNCTION Ntoken→	**integer procedure** *Ntoken*(Av_1,
←(Av_1,Nv_2,Nv_3,*c*,→	Nv_2,Nv_3,*c*,*Amatrix*,*Nmatrix1*,
←*Amatrix*,*Nmatrix1*,→	*Nmatrix2*);
←*Nmatrix2*)	**value** Av_1,Nv_2,Nv_3;
COMMON . . , . . . , . .	**real** Av_1;
DIMENSION Amatrix→	**integer** Nv_2,Nv_3;
←(*Nc*),*Nmatrix1*(Nc_1),→	**integer array** *Nmatrix1*,
←*Nmatrix2*(Nc_2)	*Nmatrix2*;
. . .	**array** *Amatrix*;
. . .	**begin**
. . .	. . .
RETURN	. . .
END	. . .
	end;

N27 N23,N24,N25

FIV:

. . . .*GE*. . . .	. . . ⩾ . . . ;

GO TO n	**go to** *n*;
	$n \leqslant \zeta$

FIV:

GO TO Nv,(n_1,n_2,→ ←. . . , n_k)	**go to** *Nv;*

This statement is preceded by appropriate *ASSIGN* declarations.

GO TO(n_1,n_2, . . . , n_k),Nv	**switch** *jump:* = n_1,n_2, . . . , n_k;
. . .	. . .
. . .	. . .
. . .	**go to** *jump*[*Nv*];
	. . .
	. . .
n_1| . . .	. . .
. . .	n_1: . . .
	. . .
n_2| . . .	
. . .	n_2: . . .
.	. . .
.	.
.	.
n_k| . . .	.
. . .	n_k: . . .
	. . .
	jump must be defined in a preceding switch declaration.

FIV:

. . . *.GT.* . . .	. . . > . . . ;

. . . *IABS(a)* . . .	. . . *entier(abs(a))* . . . ;

a:N12

IF(a) n_1,n_2,n_3	**if** $a = 0$ **then go to** n_2 **else if** $a < 0$ **then go to** n_1 **else go to** n_3**;**
*a:*N12	N17,N18
IF(a) n_1,n_1,n_2	**if** $a \leqslant 0$ **then go to** n_1 **else go to** n_2**;**
*a:*N12	N17,N18
IF(a) n_1,n_2,n_1	**if** $a = 0$ **then go to** n_2 **else go to** n_1**;**
*a:*N12	N17,N18
IF(a) n_1,n_2,n_2	**if** $a \geqslant 0$ **then go to** n_2 **else go to** n_1**;**
*a:*N12	N17,N18
FIV:	
IF(Lterm)	. . . **if** *Lterm* **then** . . . **else** . . . **;**
IF(SENSE LIGHT→ ←Nc) n_1,n_2	Control is transferred to statement n_1 or n_2 if the designated sense light on the computer console is on or off respectively. Omit in translation.

IF(SENSE SWITCH→ *←Nc)* n_1,n_2	Control is transferred to the statement whose label is n_1 or n_2 if sense switch *Nc* is on or off, respectively. The sense switches are set manually from the computer console. This statement can be used for operator controlled branches.*
. . . *IFIX(Aa)* . . .	. . . *entier(Aa)* . . . ;
FIV: *INTEGER* v_1,v_2, . . . , v_k	**integer** v_1,v_2, . . . , v_k;
N30	
FIV: *INTEGER . . , . . ,→* *←token1, . . , . .*	This statement indicates that the results of the arithmetic statement function *token1* or of the subprogram *token1* are of integer type. Omit in translation, but watch that the appropriate arithmetic statement function or the subprogram is translated by **integer procedure** *token1*. If *token1* is the identifier of a library function, then *token1* must be replaced by *entier(token1)* throughout the program, see example:

* This statement could be translated in the following way:

read Nc_1;
if $Nc_1 = 1$ **then go to** n_1 **else go to** n_2;

$Nc_1 = 1$ for SENSE SWITCH Nc on and $Nc_1 = 0$ for SENSE SWITCH Nc off.

example:

INTEGER ABS	
. . .	. . .
. . .	. . .
. . .	. . .
*K = ABS(T)*S*	*K: = entier(abs(T)) × S;*
. . .	. . .
. . .	. . .
. . .	. . .

FIV:

INTEGER→	The same as for *FUNCTION*
←FUNCTION token→	*Ntoken*(. . , . . , c_1, . . . , c_k, . . , . .)
←(. . , . . , c_1, . . . , c_k, . . ,→	only
←. .)	*Ntoken = token.*
COMMON . . , . . . , . .	
. . .	
. . .	N29
. . .	
RETURN	
END	

N27

FIV:

INTEGER→	The same as for *FUNCTION*
←FUNCTION token→	*Ntoken*(. . , . . , v_1, . . . , v_k, . . , . .)
←(. . , . . , v_1, . . . , v_k, . . ,→	only
←. .)	*Ntoken = token.*
COMMON . . , . . . , . .	
. . .	
. . .	N24,N25,N29
. . .	
RETURN	
END	

N27

FIV:

INTEGER→	The same as for *FUNCTION*
←FUNCTION token→	*Ntoken*(. . , . . , *Nmatrix*, . . , . .)
←(. . , . . , *Nmatrix*, . . ,→	only *Ntoken* = *token*.
←. .)	
COMMON . . , . . . , . .	
DIMENSION Nmatrix→	N29
←(*Nc*)	
. . .	
. . .	
. . .	
RETURN	
END	

N27

FIV:

INTEGER→	**integer procedure** *token*(. . , . . ,
←FUNCTION token→	*'text'*, . . , . .);
←(. . , . . , *cHtext*, . . , . .)	. . .
COMMON . . , . . . , . .	**string** *text;*
. . .	**begin**
. . .	. . .
. . .	. . .
RETURN	. . .
END	**end**;

N27 N29

FIV:

INTEGER→	**integer procedure** *token*(. . . . . ,
←FUNCTION token→	*token1*, . . , . .);
←(. . , . . , *token1*, . . , . .)	. . .
COMMON . . , . . . , . .	**real procedure** *token1;*

. . .	**begin**
. . .	. . .
. . .	. . .
RETURN	. . .
END	**end**;

N27

N29

The indicator **real procedure** must be replaced by **integer procedure** or by **procedure** if *token1* is the identifier of an **integer procedure** or of a **procedure**.

FIV:

INTEGER→	The same as for *FUNCTION*
←*FUNCTION token*→	*Ntoken* (Av_1,Nv_2,Nv_3,*c*,*Amatrix*,
←(Av_1,Nv_2,Nv_3,*c*,→	*Nmatrix1*,*Nmatrix2*) only
←*Amatrix*,*Nmatrix1*,→	*Ntoken* = *token*.
←*Nmatrix2*)	
COMMON . . , . . . , . .	
DIMENSION Amatrix→	N24,N25,N29
←(*Nc*), $Nmatrix_1(Nc_1)$,→	
←$Nmatrix2(Nc_2)$	
. . .	
. . .	
. . .	
RETURN	
END	

N27

FIV:

. . . *ISIGN*(*Na*) . . .	. . . *sign*(*Na*) . . . ;

FIV:

. . . .*LE*. . . .	. . . ⩽ . . . ;

. . . *LOG*(*a*) . . .	. . . *ln*(*a*) . . . ;

a:N12

. . . *LOGF*(*a*) . . .	. . . *ln*(*a*) . . . ;

a:N12,N42

FIV:	
LOGICAL $v_1,v_2, \ldots, v_k$	**Boolean** $v_1,v_2, \ldots, v_k$;

FIV:	
LOGICAL . . , token1, . .	This statement indicates that *token1* is a logical statement function or that the subprogram *token1* is of logical type. Omit in translation, but watch that the appropriate arithmetic statement function or subprogram is translated by **Boolean procedure** *token1*.

FIV:	
LOGICAL→	**Boolean procedure** *token*(. . , . . ,
←*FUNCTION token*→	$c_1, \ldots, c_k, \ldots, .$)·
←(. . , . . , $c_1, \ldots, c_k$, . . ,→	. . .
←. .)	**begin**
COMMON . . , . . , . .	. . .
. . .	. . .
. . .	. . .
. . .	**end**;
RETURN	
END	
	N29

N27

FIV:

LOGICAL→	**Boolean procedure** *token*(. . , . . ,
←FUNCTION token→	$v_1, \ldots, v_k, \ldots, \ldots$);
←(. . , . . , $v_1, \ldots, v_k, \ldots$,→	**value** . . , $v_1, \ldots, v_k, \ldots$;
←. .)	**real** . . , . . . , . . ;
COMMON . . , . . . , . .	**integer** . . , . . . , . . ;
. . .	**Boolean** . . , . . . , . . ;
. . .	. . .
. . .	**begin**
RETURN	. . .
END	. . .
	. . .
	end;

N27

N24,N29,N36

FIV:

LOGICAL→	**Boolean procedure** *token*(. . , . . ,
←FUNCTION token→	*Nmatrix*, . . , . .);
←(. . , . . , *Nmatrix*, . . ,→	. . .
←. .)	**integer array** *Nmatrix*;
COMMON . . , . . . , . .	. . .
DIMENSION Nmatrix→	**begin**
←(*Nc*)	. . .
. . .	. . .
. . .	. . .
. . .	**end**;
RETURN	
END	
	N29

N27

FIV:

LOGICAL→	**Boolean procedure** *token*(. . , . . ,
←FUNCTION token→	*token1*, . . , . .);
←(. . , . . , *token1*, . . , . .)	. . .
COMMON . . , . . , . .	**real procedure** *token1;*
. . .	**begin**
. . .	. . .
. . .	. . .
RETURN	. . .
END	**end**;

N27

N29

The indicator **real procedure** must be replaced by **integer procedure** or by **Boolean procedure** or by **procedure** if *token1* is the identifier of an **integer procedure** or of a **Boolean procedure** or of a **procedure**.

FIV:

LOGICAL→	**Boolean procedure** *token*(Av_1,Nv_2,
←FUNCTION token→	Nv_3,*c*,*Amatrix*,*Nmatrix1*,
←(Av_1,Nv_2,Nv_3,*c*,*→*	*Nmatrix2*,*v*);
←Amatrix,*Nmatrix1*,*→*	**value** Av_1,Nv_2,Nv_3,*v;*
←Nmatrix2,*v*)	**real** Av_1;
COMMON . . , . . , . .	**integer** Nv_2,Nv_3;
DIMENSION→	**Boolean** *v;*
←Amatrix(*Nc*),*→*	**integer array** *Nmatrix1*,
←Nmatrix1(Nc_1),*→*	*Nmatrix2;*
←Nmatrix2(Nc_2)	**array** *Amatrix;*
. . .	**begin**
. . .	. . .
. . .	. . .
RETURN	. . .
END	**end**;

N27

N24,N29,N36

---

logical operators:	
.OR.	$\vee$
.AND.	$\wedge$
.NOT.	$\neg$
logical values:	
.TRUE.	**true**
.FALSE.	**false**
FIV:	
. . . *.LT.* . . .	. . . < . . .;
multiplication:	
a_1*a_2	$a_1 \times a_2$
FIV:	
. . . *.NE.* . . .	. . . ≠ . . .;
FIV:	
. . . *.NOT.* . . .	. . . ¬ . . .;

numbers:

0	0
0.	0
123	123
123.	123
.1234	.1234
+0.1234	+0.1234
−123.456	−123.456
+1.23*E*5	$+1.23_{10}5$
9.87*E*+12	$9.87_{10}+12$
2.E−3	$2_{10}-3$
8E4	$8_{10}4$
−0.012E−03	$-0.012_{10}-03$
−1.E8	$-_{10}8$
1.E−5	$_{10}-5$
+1.E+5	$+_{10}+5$

Range for real numbers: ϵ
Range for integer numbers: δ
Range for exponents: η

The following numbers appear only in connection with input data:

E2	$_{10}2$
E02	$_{10}2$
+2	$_{10}2$
−2	$_{10}-2$
+02	$_{10}2$

FIV:
double-precision numbers:

21.9D0	21.9
0.203D0	0.203
8.0D3	$8.0_{10}3$
5.5D − 5	$5.5_{10} - 5$

complex numbers:

(7.3,1.76) is equal to 7.3 + 1.76i
(5.1E2, − 3.23) is equal to 510 − 3.23i
(1.1,0.0) is equal to 1.1 + 0.0i

FIV:

. . . *.OR.* . . .	. . . *V* . . . ;

PAUSE	This statement halts the computer. Pressing the start key causes the program to resume execution of the program with the next statement.
or	
PAUSE n	In a *PAUSE n* statement, the number *n* is displayed on the console during the halt.

FIV:

PRINT n,v	This statement is used to print out *v* on the line-printer in accordance with the format statement *n*. This format statement *n* includes information to space the printed output properly.

PUNCH n

This statement causes information to be punched out according to the format statement *n*.

PUNCH n,v

N8

This statement causes the value of the variable identifier *v* to be punched out according to the format statement *n*.

NE 803:
PRINT { },*v*;

N15

PUNCH n,$v_1, \ldots, v_k$

N8

This statement causes the values of the variable identifiers $v_1, \ldots, v_k$ to be punched out according to the format statement *n*.

NE 803:
PRINT { },$v_1, \ldots, v_k$;

N15

PUNCH n,(*matrix*(*Nv*),→
←*Nv* = Na_1,Na_2)

N8

for *Nv*: = Na_1 **step** *1* **until** Na_2 **do** output statement for *matrix* [*Nv*];
Nv,Na_1,Na_2:N14

NE 803:
FOR Nv: = Na_1 *STEP 1 UNTIL* Na_2 *DO PRINT* { },
matrix(*Nv*);

N15

PUNCH n,(matrix(Nv),→
←Nv = Na1,Na2,Na3)

for *Nv*: = *Na*1 **step** *Na*3 **until** *Na*2 **do** output statement for *matrix*[*Nv*];

*Nv,Na*1,*Na*2,*Na*3:N14

N8

NE 803:
FOR Nv: = *Na*1 *STEP Na*3
*UNTIL Na*2 *DO PRINT* { },
matrix(*Nv*);

N15

PUNCH n,((matrix→
*←(Nv*1,*Nv*2),*Nv*2 = *1*,→
*←m*2),*Nv*1 = *1*,*m*1)

for *Nv*2: = *1* **step** *1* **until** *m*2 **do**
for *Nv*1: = *1* **step** *1* **until** *m*1 **do**
output statement for *matrix*[*Nv*1, *Nv*2];

*Nv*1,*Nv*2,*m*1,*m*2:N14

N8

NE 803:
*FOR Nv*2: = *1 STEP 1 UNTIL*
*m*2 *DO*
*FOR Nv*1: = *1 STEP 1 UNTIL*
*m*1 *DO*
PRINT { },*matrix*(*Nv*1,*Nv*2);

N15

PUNCH n,((matrix(Nv₁,→
←Nv₂),Nv₂ = m₁,m₂),→
←Nv₁ = m₃,m₄)

for *Nv₂: = m₁* **step** *1* **until** *m₂* **do**
for *Nv₁: = m₃* **step** *1* **until** *m₄* **do**
output statement for *matrix*[*Nv₁*, *Nv₂*];

Nv₁,Nv₂,m₁,m₂,m₃,m₄ :N14

N8

NE 803:
FOR Nv₂: = m₁ STEP 1 UNTIL m₂ DO
FOR Nv₁: = m₃ STEP 1 UNTIL m₄ DO
PRINT { },*matrix*(*Nv₁*,*Nv₂*);

N15

PUNCH n,((matrix→
←(Nv₁,Nv₂),Nv₂ = Na₁,→
←Na₂,Na₃),Nv₁ = Na₄,→
←Na₅,Na₆)

for *Nv₂: = Na₁* **step** *Na₃* **until** *Na₂*
do for *Nv₁: = Na₄* **step** *Na₆* **until**
Na₅ **do** output statement for
matrix[*Nv₁*,*Nv₂*];

Nv₁,Nv₂,Na₁, . . . , Na₆:N14

N8

NE 803:
FOR Nv₂: = Na₁ STEP Na₃ UNTIL Na₂ DO FOR Nv₁: = Na₄ STEP Na₆ UNTIL Na₅ DO PRINT{ },*matrix*(*Nv₁*,*Nv₂*);

N15

PUNCH n,(matrix→ *←(Nv2),matrix1(Nv3,→* *←Nv4),Nv2 = Na1,Na2,→* *←Na3)*	**for** *Nv2: = Na1* **step** *Na3* **until** *Na2* **do** output statement for *matrix*[*Nv2*],*matrix1*[*Nv3*,*Nv4*]; *Nv2,Nv3,Nv4,Na1,Na2,Na3:*N14
N8	NE 803: *FOR Nv2: = Na1 STEP Na3 UNTIL Na2 DO PRINT* { }, *matrix*(*Nv2*),*matrix1*(*Nv3*,*Nv4*);
	N15

FIV:

PUNCH TAPE n,v	This statement causes *v* to be punched by the paper tape punch according to the format statement *n*.

READ n,matrix	This statement causes the values of the array identifier *matrix* to be read in according to the format statement *n*.
N8	
example:	NE 803:
DIMENSION NUT(5,8) . . . . . . *n*| *FORMAT*({ }) *READ n,NUT*	*INTEGER ARRAY NUT(1:5, 1:8);* . . . . . . *FOR N: = 1 STEP 1 UNTIL 8 DO FOR M: = 1 STEP 1 UNTIL 5 DO READ NUT*(*M*, *N*); *N,M:*N14

	READ n	This statement causes information to be read in according to the format statement *n*.
example:		
n|	*FORMAT(cHtext)* *READ n*	NE 803: *READ $ text ?;*
	READ n,a	This statement causes the value of the identifier *a* to be read in according to the format statement *n*.
N8		
		NE 803: *READ a;*
		a:N14
	READ n,a$_1$, . . . , *a*$_k$	This statement causes the values of the identifiers *a*$_1$, . . . , *a*$_k$ to be read in according to the format statement *n*.
N8		
		NE 803: *READ a*$_1$, . . . , *a*$_k$*;*
		a$_1$, . . . , *a*$_k$:N14
FIV:		
	READ(Nc) a$_1$, . . . , *a*$_k$	This statement causes *a*$_1$, . . . , *a*$_k$ to be read in in binary form from input device *Nc*.

FIV:

READ(Nc,n) $a_1, \ldots, a_k$

This statement causes $a_1, \ldots, a_k$ to be read in from input device *Nc* according to format statement *n*.

READ n,(matrix(Nv),→
←Nv = Na1,Na2)

for *Nv:* = Na_1 **step** *1* **until** Na_2 **do** input statement for *matrix* [*Nv*];

Nv,Na_1,Na_2:N14

N8

NE 803:
FOR Nv: = Na_1 *STEP 1 UNTIL* Na_2 *DO READ matrix (Nv);*

READ n,(matrix(Nv),→
←Nv = Na1,Na2,Na3)

for *Nv:* = Na_1 **step** Na_3 **until** Na_2 **do** input statement for *matrix* [*Nv*];

Nv,Na_1,Na_2,Na_3:N14

N8

NE 803:
FOR Nv: = Na_1 *STEP* Na_3 *UNTIL* Na_2 *DO READ matrix (Nv);*

READ n,((matrix(Nv1,→
←Nv2),Nv2 = 1,m2),→
←Nv1 = 1,m1)

for Nv_2: = *1* **step** *1* **until** m_2 **do** **for** Nv_1: = *1* **step** *1* **until** m_1 **do** input statement for *matrix*[Nv_1, Nv_2];

Nv_1,Nv_2,m_1,m_2:N14

N8

NE 803:
FOR Nv_2: = *1 STEP 1 UNTIL* m_2 *DO FOR* Nv_1: = *1 STEP 1 UNTIL* m_1 *DO READ matrix* (Nv_1,Nv_2);

READ n,((matrix→
←(Nv1,Nv2),Nv2 = m1,→
←m2),Nv1 = m3,m4)

for *Nv2:* = *m1* **step** *1* **until** *m2* **do**
for *Nv1:* = *m3* **step** *1* **until** *m4* **do**
input statement for *matrix*[*Nv1*, *Nv2*];

*Nv1,Nv2,m1, . . . , m4:*N14

N8

NE 803:
FOR Nv2: = *m1 STEP 1 UNTIL m2 DO FOR Nv1:* = *m3 STEP 1 UNTIL m4 DO READ matrix (Nv1,Nv2);*

READ n,((matrix→
←(Nv1,Nv2),Nv2 = Na1,→
←Na2,Na3),Nv1 = Na4,→
←Na5,Na6)

for *Nv2:* = *Na1* **step** *Na3* **until** *Na2* **do for** *Nv1:* = *Na4* **step** *Na6* **until** *Na5* **do** input statement for *matrix* [*Nv1,Nv2*];

*Nv1,Nv2,Na1, . . . , Na6:*N14

N8

NE 803:
FOR Nv2: = *Na1 STEP Na3 UNTIL Na2 DO FOR Nv1:* = *Na4 STEP Na6 UNTIL Na5 DO READ matrix(Nv1,Nv2);*

READ n,(matrix(Nv2),→
←matrix1(Nv3,Nv4),→
←Nv2 = Na1,Na2,Na3)

for *Nv2:* = *Na1* **step** *Na3* **until** *Na2* **do** input statement for *matrix* [*Nv2*],*matrix1*[*Nv3,Nv4*];

*Nv2,Nv3,Nv4,Na1,Na2,Na3:*N14

N8

NE 803:
FOR Nv2: = *Na1 STEP Na3 UNTIL Na2 DO READ matrix (Nv2),matrix1(Nv3,Nv4);*

FIV:

READ INPUT TAPE→
←Nc,n,a

This statement causes the value of the identifier *a* to be read in from the magnetic tape unit *Nc* according to the format statement *n*.

FIV:

READ TAPE Nc,a

This statement causes the value of the identifier *a* to be read in from the magnetic tape unit *Nc*.

FIV:

REAL $v_1, \ldots, v_k$

real $v_1, \ldots, v_k$;

N30

FIV:

REAL . . , . . , token1,→
←. . , . .

This statement indicates that the result of the arithmetic statement function *token1*, or of the library function *token1*, or of the subprogram *token1* are of real type. Omit in translation, but watch that the appropriate arithmetic statement function or the subprogram is translated by **real procedure** *token1*.

FIV:

REAL FUNCTION→
←token(. . , . . , $c_1, \ldots,$*→*
←c_k*, . . , . .)*
COMMON . . , . . , . .
. . .
. . .
. . .
RETURN
END

The same as for *FUNCTION Atoken(. . , . . ,* $c_1, \ldots,$ c_k*, ...)* only *Atoken = token*.

N27

FIV:

REAL FUNCTION→	The same as for *FUNCTION*
←*token*(. . , . . , v_1, . . . ,→	*Atoken*(. . , . . , v_1, . . . , v_k, . . , . .)
←v_k, . . , . .)	only *Atoken* = *token*.
COMMON . . , ... , . .	
. . .	
. . .	N24,N25,N29
. . .	
RETURN	
END	

N27

FIV:

REAL FUNCTION→	The same as for *FUNCTION*
←*token*(. . , . . ,→	*Atoken*(. . , . . , *Nmatrix*, . . , . .)
←*Nmatrix*,. . , . .)	only *Atoken* = *token*.
COMMON . . , ... , . .	
DIMENSION Nmatrix→	
←(*Nc*)	N29
. . .	
. . .	
. . .	
RETURN	
END	

N27

FIV:

REAL FUNCTION→	**real procedure** *token*(. . , . . ,
←*token*(. . , . . , *cHtext*,→	'*text*', . . , . .);
←. . , . .)	. . .
COMMON. . , ... , . .	**string** *text*;
. . .	**begin**
. . .	. . .
. . .	. . .
RETURN	. . .
END	**end**;

N27 N29

FIV:

REAL FUNCTION→	**real procedure** *token*(. . , . . ,
←*token*(. . , . . , *token1*,→	*token1*, . . , . .);
←. . , . .)	. . .
COMMON . . , . . . , . .	**real procedure** *token1;*
. . .	**begin**
. . .	. . .
. . .	. . .
RETURN	. . .
END	**end**;

N27

N23
The indicator **real procedure** must be replaced by

integer procedure
Boolean procedure
procedure

if *token1* is the identifier of an

integer procedure
Boolean procedure
procedure

FIV:

REAL FUNCTION→	The same as for *FUNCTION*
←*token*(Av_1,Av_2,→	*token*(Av_1,Av_2,Nc_3,c_1,c_2,*Amatrix*)
←Nc_3,c_1,c_2,*Amatrix*)	only *Atoken* = *token*.
COMMON . . , . . . , . .	
DIMENSION Amatrix→	
←(*Nc*)	N24,N25,N29
. . .	
. . .	
. . .	
RETURN	
END	

N27

RECORD(*a*) $a_1, \ldots, a_k$	This statement is used to write $a_1, \ldots, a_k$ into the disk storage on the record *a*.

relational operators:

.LT.	$<$
.LE.	$\leqslant$
.EQ.	$=$
.GE.	$\geqslant$
.GT.	$>$
.NE.	$\neq$

RETURN	**go to** *a;*
The statement directly following is not *END.*	The last **end** statement of the subprogram must be replaced by *a:***end;**

example:

FUNCTION KLAUS(*T*)	**integer procedure** *KLAUS*(*T*)*;*
	. . .
	. . .
. . .	. . .
RETURN	**go to** *LOS;*
	. . .
. . .	. . .
RETURN	*LOS:* **end;**
END	

RETURN	**end;**
END	

FIV:

REWIND Nc	This statement causes an end-of-file mark to be written on the tape of magnetic tape unit *Nc* and the tape to be rewound.
SENSE LIGHT Nc	If $Nc = 0$, all sense lights on the control console will be turned off, otherwise the sense light specified by *Nc* will be turned on.

FIV:

. . . *SIGN*(*Aa*) . . .	. . . *sign*(*Aa*) . . . ;

FIV, FII:

. . . *SIN*(*a*) . . .	. . . *sin*(*a*) . . . ;

a:N12

. . . *SINF*(*a*) . . .	. . . *sin*(*a*) . . . ;

a:N12, N42

FIV, FII:

. . . *SQRT*(*a*) . . .	. . . *sqrt*(*a*) . . . ;

a:N12

. . . *SQRTF*(*a*) . . .	. . . *sqrt*(*a*) . . . ;

a:N12,N42

statement label (number):

n| . . .	*n*: . . . ;

statement separator:

No special statement separator, different statements must be on different records.	; The statements are separated by a semicolon.

STOP or *STOP n*	This statement causes a halt in such a way that pressing the start key has no effect. The message *STOP* is typed on the console typewriter. If *STOP n* is used, the number *n* is displayed on the console.

SUBROUTINE token	**procedure** *token;*
COMMON . . , . . . , . .	**begin**
. . .	. . .
. . .	. . .
. . .	. . .
RETURN	**end**;
END	
	N23
N27	

SUBROUTINE token→	**procedure** *token*(. . , . . , c_1, . . . ,
←(. . , . . , c_1, . . . , c_k . . , . .)	c_k, . . , . .);
COMMON . . , . . . , . .	. . .
. . .	**begin**
. . .	. . .
. . .	. . .
RETURN	. . .
END	**end**;
N27	N23

SUBROUTINE token→	**procedure** *token*(. . , . . , v_1, . . . ,
←(. . , . . , v_1, . . . , v_k, . . ,→	v_k, . . , . .);
←. .)	**value** . . , v_1, . . . , v_k, . . ;
COMMON . . , . . . , . .	**real** . . , . . . , . . ;
. . .	**integer** . . , . . . , . . ;
. . .	. . .
. . .	**begin**
RETURN	. . .
END	. . .
	. . .
	end;
N27	
	N23,N24,N25

SUBROUTINE token→	**procedure** *token*(. . , . . , *n*, . . , . .);
←(. . , . . , *n*, . . , . .)	. . .
COMMON . . , . . , . .	**label** *n;*
. . .	**begin**
. . .	. . .
. . .	. . .
RETURN	. . .
END	**end**;

N27 N23

n is a statement label (number).

SUBROUTINE token→	**procedure** *token*(. . , . . , *Nmatrix*,
←(. . , . . , *Nmatrix*, . . ,→	. . , . .);
←. .)	. . .
COMMON . . , . . , . .	**integer array** *Nmatrix;*
DIMENSION Nmatrix(*Nc*)	**begin**
. . .	. . .
. . .	. . .
. . .	. . .
RETURN	**end**;
END	

N23

N27

SUBROUTINE token→	**procedure** *token*(. . , . . , *'text'*, . . ,
←(. . , . . , *cHtext*, . . , . .)	. .);
COMMON. . , . . , . .	. . .
. . .	**string** *text;*
. . .	**begin**
. . .	. . .
RETURN	. . .
END	. . .
	end;

N27

N23

FIV:

SUBROUTINE token→	**procedure** *token*(. . , . . , *token1*, . . ,
←(. . , . . , $token_1$, . . , . .)	.);*
COMMON . . , . . . , . .	. . .
. . .	**real procedure** *token1;*
. . .	**begin**
. . .	. . .
RETURN	. . .
END	. . .
	end;

N27

N23
The indicator **real procedure** must be replaced by **integer procedure** or **procedure** if *token1* is the identifier of an **integer procedure** or of a **procedure**.

SUBROUTINE token→	**procedure** *token*(*Av*,Av_1,*Nmatrix1*,
←(*Av*,Av_1,*Nmatrix1*,→	*Amatrix2*,*Amatrix3*,*term*);
←*Amatrix2*,*Amatrix3*,→	**value** *Av*,Av_1;
←*term*)	**real** *Av*,Av_1;
COMMON . . , . . . , . .	**integer array** *Nmatrix1;*
DIMENSION Nmatrix1→	**array** *Amatrix2*,*Amatrix3;*
←(*Nc*)	**begin**
DIMENSION Amatrix2→	. . .
←(Nc_1)	. . .
DIMENSION Amatrix3→	. . .
←(Nc_2)	**end**;
. . .	
. . .	
. . .	N23,N24,N25,N26
. . .	
RETURN	
END	

N27

SUBROUTINE token→ ←(*Nv*,*Av*$_1$,*Av*$_2$,*c*,*c*$_1$,*term*,→ ←*n*$_1$,*n*$_2$) *COMMON* . . , . . . , ... *RETURN* *END*	**procedure** *token*(*Nv*,*Av*$_1$,*Av*$_2$,*c*,*c*$_1$, *term*,*n*$_1$,*n*$_2$); **value** *Nv*,*Av*$_1$,*Av*$_2$; **real** *Av*$_1$,*Av*$_2$; **integer** *Nv*; **label** *n*$_1$,*n*$_2$; **begin** . . . . . . **end**;
N27	N23,N24,N25

subtraction:

$a_1 - a_2$	$a_1 - a_2$

FIV:

. . . *token*(. . , . . . , . .) . . .	. . . *token*(. . , . . . , . .) . . .;
token is the identifier of an *INTEGER FUNCTION* or of a *REAL FUNCTION* or of a *LOGICAL FUNCTION* or of an arithmetic statement function.	The declaration of the appropriate **integer procedure** or **real procedure** or **Boolean procedure** must precede this statement.

FIV:

. . . *token*(. . , . . , *token1*,→ ←. . , . .) . . .	. . . *token*(. . , . . , *token1*, ...) . . .;
token is the identifier of a *REAL FUNCTION* or of an *INTEGER FUNCTION* or of a *LOGICAL FUNCTION*.	The declaration of the appropriate **real procedure** or **integer procedure** or **Boolean procedure** must precede this statement.

. . . *Atoken*(. . , . . . , . .) . . .	. . . *Atoken*(. . , . . . , . .) . . . ;
Atoken is the identifier of a *FUNCTION* or of an arithmetic statement function.	The declaration of the appropriate **real procedure** must precede this statement.

. . . *Ntoken*(. . , . . . , . .) . . .	. . . *Ntoken*(. . , . . . , . .) . . . ;
Ntoken is the identifier of a *FUNCTION* or of an arithmetic statement function.	The declaration of the appropriate **integer procedure** must precede this statement.
. . . *AtokenF*(. . . . . .)→ ←. . .	. . . *AtokenF*(. . , . . . , . .) . . . ;
AtokenF is the identifier of an arithmetic statement function.	The declaration of the appropriate **real procedure** must precede this statement.
. . . *NtokenF*(. . . . . .)→ ←. . .	. . . *NtokenF*(. . , . . . , . .) . . . ;
Ntoken is the identifier of an arithmetic statement function.	The declaration of the appropriate **integer procedure** must precede this statement.
. . . *XtokenF*(. . , . . . , . .)→ ←. . .	. . . *XtokenF*(. . , . . . , . .) . . . ;
XtokenF is the identifier of an arithmetic statement function.	The declaration of the appropriate **integer procedure** must precede this statement.
FIV: *token*(. . . . . .) = *term* arithmetic statement function.	**real procedure** *token*(. . , . . . , . .); **value** . . , . . . , . . ; **real** . . , . . . , . . ; **integer** . . , . . . , . . ; **begin** *token*: = *term*; **end**; N24,N25 It is a **real procedure** if *token* is declared in the program as real, and an **integer procedure** if *token* is declared as integer. The identifiers of *term* must be declared in the main program.

Atoken(. . , . . . , . .) = *term*

arithmetic statement function.

real procedure *Atoken*(. . , . . . , . .);
value . . , ... ;
real . . . . . ;
integer . . . . . ;
begin
Atoken : = *term*;
end;

N24,N25
The identifiers of *term* must be declared in the main program.

Ntoken(. . , . . . , . .) = *term*

arithmetic statement function.

integer procedure *Ntoken*(. . . . . , . .);
value . . , ... , . . ;
real . . , . . . , . . ;
integer . . , . . . , . . ;
begin
Ntoken : = *term*:
end;

N24,N25
The identifiers of *term* must be declared in the main program.

AtokenF(. . , . . . , . .) =→ ←*term*

arithmetic statement function.

real procedure *AtokenF*(. . , . . . , . .);
value . . , ... ;
real . . . . . ;
integer . . , ...
begin
AtokenF : = *term*;
end;

N24,N25
The identifiers of *term* must be declared in the main program.

NtokenF(. . . . . , . .) =→ ←*term* arithmetic statement function.	**integer procedure** *NtokenF*(. . , . . . , . .); **value** . . . . . ; **real** . . . . . ; **integer** . . , ... , . . ; **begin** *NtokenF* : = *term*; **end**; N24,N25 The identifiers of *term* must be declared in the main program.
XtokenF(. . . . . , . .) =→ ←*term* arithmetic statement function.	**integer procedure** *XtokenF*(. . , . . . , . .); **value** . . , . . . , . . ; **real** . . , . . . , . . ; **integer** . . , . . . , . . ; **begin** *XtokenF*: = *term*; **end**; N24,N25 The identifiers of *term* must be declared in the main program.
FIV: . . . . *TRUE*. . . .	. . . **true** . . . ;
TYPE n,a	This statement is used to type out the value of the variable identifier *a* on the console typewriter according to the format statement *n*.

FIV:
type declaration:

See *INTEGER . . , . . . , . .*
or *REAL . . , . . . , . .*
or *LOGICAL . . , . . . , . .*

(integer) variable identifier:

Not more than α characters are used to identify a variable identifier.
Those variable identifiers must be listed in the type declaration under **integer**.

examples:

K	*K*
MG42	*MG42*
NE803	*NE803*
J1A2	*J1A2*
LANDAU	*LANDAU*

(real) variable identifiers:

Not more than α characters are used to identify a variable identifier.
Those variable identifiers must be listed in the type declaration under **real**.

examples:

E605	*E605*
AX121	*AX121*
B12	*B12*
DDT	*DDT*
Z51AAA	*Z51AAA*

(subscripted) variable identifier:

Not more than α characters are used to identify a variable identifier.
The subscript appears in square brackets.
It is not necessary to list those variable identifiers in a type declaration.
The maximum value for a subscript is β.
N9

examples:

X(1)	*X[1]*
HANS(38)	*HANS[38]*
BETA(I,N)	*BETA[I,N]*
KALLE(8,M,105)	*KALLE[8,M,105]*
M30(I)	*M30[I]*
A1(K + 1,L)	*A1[K + 1,L]*
*DELTA(3*M − 5)*	*DELTA[3 × M − 5]*

FIV:

WRITE(Nc) $a_1, \ldots, a_k$

This statement causes $a_1, \ldots, a_k$ to be written, in binary form, on the device *Nc*.

FIV:

WRITE(Nc,n) $a_1, \ldots, a_k$

This statement causes $a_1, \ldots, a_k$ to be written on device *Nc* in accordance with format statement *n*.

FIV:

WRITE OUTPUT→
←TAPE Nc,n,a

This statement causes *a* to be written on the magnetic tape unit *Nc* in accordance with format statement *n*.

FIV:

WRITE TAPE Nc,a	This statement causes *a* to be written, in binary form, on the magnetic tape unit *Nc*.

. . . *XABSF*(*a*) . . .	. . . *entier*(*abs*(*a*)) . . . ;

a:N12

. . . *XEXPF*(*a*) . . .	. . . *entier*(*exp*(*a*)) . . . ;

a:N12

. . . *XLOGF*(*a*) . . .	. . . *entier*(*ln*(*a*)) . . . ;

a:N12

. . . *XSQRTF*(*a*) . . .	. . . *entier*(*sqrt*(*a*)) . . . ;

a:N12

. . . *Xtoken*(. . , . . . , . .) . . .	. . . *Xtoken*(. . , . . . , . .) . . . ;
Xtoken is the identifier of an arithmetic statement function.	The declaration of the appropriate **integer procedure** must precede this statement.

Xtoken(. . , . . . , . .) = *term* arithmetic statement function.	The same as for *Ntoken*(. . , . . . , . .) = *term* only *Ntoken* = *Xtoken*.

Part 6

The Method of Translating a Program

6.1 ALGOL 60 INTO FORTRAN

On the following pages the method of translating an ALGOL 60 program into a FORTRAN program is described. The translation of the program itself is explained, together with a comment on how to adapt input data. The input/output statements are not treated here, they are discussed in Part 3.

Three different ALGOL 60 (6.1.4) and ALGOL (6.1.5 and 6.1.6)* programs and their translations into FORTRAN II serve as examples to illustrate the method of translation.†

The ALGOL programs are computed on the NE 803B (8K memory, paper tape input/output)‡ and the FORTRAN II programs were run on the IBM 1620_{II} (40,000 character memory, punched cards input/output).

6.1.1 Translation of the Program into FORTRAN II.

To translate an ALGOL program into a FORTRAN II program, it is recommended that the following pattern be followed:

1. Begin at the type declaration. Change the first letter of the variable identifiers declared under **integer**, so that the first letter becomes *I*, *J*, *K*, *L*, *M*, or *N*. (Naturally no change is necessary if the variable identifier already begins with *I*, *J*, *K*, *L*, *M*, or *N*.) Change the first letter of the variable identifiers declared under **real**, so that the first letter is not *I*, *J*, *K*, *L*, *M*, or *N*.

2. Replace the changed variable identifiers throughout the entire program with the new variable identifiers.

3. The identifier of a subscripted variable declared under **array** or **real array** must be changed in such a manner that the first character is not *I*, *J*, *K*, *L*, *M*, or *N*. The identifier of a subscripted variable declared under **integer array** must be changed in such a manner that the first letter is *I*, *J*, *K*, *L*, *M*, or *N*.

4. Replace the changed array identifiers throughout the entire program with the new array identifiers.

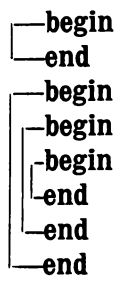
* An ALGOL program is an ALGOL 60 program which includes input/output statements.

† This includes data input and output of the computed results.

‡ The program itself contains NE input/output statements, but the ALGOL 60 reference language symbols are not altered due to the hardware representation of the NE.

5. Examine the right side of every statement throughout the entire program to see whether it contains only identifiers and numbers of the same type. Keep in mind that every number with a decimal point or written as a power of ten is of real type. If real and integer identifiers are mixed, then change the integer identifiers into real identifiers in a statement, which precedes the mixed statement. If integer numbers appear together with real identifiers, then change the integer numbers by inserting a decimal point.

6. Link the different correlated pairs of **begin** and **end** with a line. These correlated pairs can be found by linking the innermost **begin** with the next **end**, then repeat the linking by starting with the now innermost **begin**.



begin and **end** act similar to opening and closing brackets and so it is very useful to see which of these pairs belong together.

7. If the program contains **integer procedure** and/or **real procedure** and/or **procedure**, then translate these subprograms separately. (Unlike ALGOL, a FORTRAN subprogram is generally compiled separately from the main program.) By comparing the correlated **begin** and **end** pairs, one finds the end of a subprogram very easily. The translation of these subprograms must be carried out in the same way as for a new program, i.e. one should start with the same pattern as given above.

8. Watch carefully whether a statement beginning with **if** is an **if** . . . **then** or an **if** . . . **then** . . . **else** statement.

9. Now translate statement by statement, beginning with the very first statement of the program, with the aid of Part 4.

10. When you reach input and/or output statements, see Part 3.

11. Do not hesitate to use *CONTINUE* statements; it makes the program clearer.

12. After translating the given program statement by statement, one may, if one wishes, rearrange the sequence of some records in order to

get a clearer program (this rearrangement is not carried out in the example of Part 6). The recommended sequence of statements is:

DIMENSION
EQUIVALENCE
COMMON
FORMAT
main program (includes comment cards)
subprograms

6.1.2 Translation of the Program into FORTRAN IV

To translate an ALGOL 60 program into a FORTRAN IV program, it is recommended that the following pattern be followed:

1. Although it is optional to use the type declaration, it is highly recommended that the type declaration be translated according to Part 4.

2. The identifier of a subscripted variable declared under **array** or **real array** must be changed in such a manner that the first character is not *I*, *J*, *K*, *L*, *M*, or *N*. The identifier of a subscripted variable declared under **integer array** must be changed in such a manner that the first letter is *I*, *J*, *K*, *L*, *M*, or *N*.

3. Replace the changed array identifiers throughout the entire program with the new array identifiers.

4. Examine the right side of every statement throughout the entire program to see whether it contains only identifiers and numbers of the same type. Keep in mind that every number with a decimal point or written as a power of ten is of real type. If real and integer identifiers are mixed, then change the integer identifiers (by using the first letter, as in FORTRAN II, for determining the type) into real identifiers in a statement, which precedes the mixed statement. If integer numbers appear together with real identifiers, then change the integer numbers by inserting a decimal point.

5. Link the different correlated pairs of **begin** and **end** with a line. These correlated pairs can be found by linking the innermost **begin** with the next **end**, then repeat the linking by starting with the now innermost **begin**.

begin and **end** act similar to opening and closing brackets, and so it is very useful to see which of these pairs belong together.

6. If the program contains **integer procedure** and/or **real procedure** and/or **procedure**, then translate these subprograms separately. (Unlike ALGOL 60 a FORTRAN subprogram is generally compiled separately from the main program.) By comparing the correlated **begin** and **end** pairs, one finds the end of a subprogram very easily. The translation of these subprograms must be carried out in the same way as for a new program, i.e. one should start with the same pattern as given above.

7. Watch carefully whether a statement beginning with **if** is an **if** . . . **then** or an **if** . . . **then** . . . **else** statement.

8. Now translate statement by statement, beginning with the very first statement of the program, with the aid of Part 4.

9. When you reach input and/or output statements, see Part 3.

10. Do not hesitate to use *CONTINUE* statements; it makes the program clearer.

11. After translating the given program, statement by statement, one can (optionally) rearrange the sequence of some records in order to get a clearer program. (This rearrangement is not carried out in the example of Part 6.) The recommended sequence of statements is:

REAL
INTEGER
LOGICAL
DIMENSION
DATA
EQUIVALENCE
COMMON
FORMAT
main programs (includes comment records)
subprograms

6.1.3 The Data

To translate input data for an ALGOL program into input data for a FORTRAN program, one should keep the following points in mind:

1. If the ALGOL processor follows the FORTRAN input rules (see Part 3), the input data may be copied directly.

2. If the ALGOL processor does not follow the FORTRAN input rules, then the numbers or values can be separated by any character except 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, $_{10}$, ., −, or +. Most likely they are separated by using spaces (blanks), by using a new record, or by using a new line. In contrast to this, numbers or values in FORTRAN are

identified by appropriate format statements, and though it is not necessary to separate the different numbers or values on the record, certainly a separation is recommended.

3. The format of every number or value must be determined by the main program, i.e. if the variable identifier is of integer type, then a *FORMAT*(*Iw*) must be chosen, etc. Although certain relaxations in input data format are permitted, it is highly recommended that the input data be written on the record in exact agreement with the appropriate format statement. That means that a number with an appropriate *E* or *F* format statement must have a decimal point.

4. Not all FORTRAN processors accept a number without a decimal point in connection with an exponent. For that reason, it is recommended that every number in connection with an exponent be written with a decimal point, i.e. not 5E22 but 5.E22.

5. After keeping this recommendation in mind, translate the input data with the aid of Part 4 "numbers".

6.1.4 A Very Simple Example

program

```
begin comment  LANDAUVERTEILUNG NACH BLUNCK-LEISEGANG KORRIGIERT
               (ZEITSCHRIFT FUER PHYSIK 130, 641-649),
               MIT PRIMAERSPEKTRUM GEFALTET

               HORST THEISSEN DEZ 1963;


               integer N,Z,A,K;
               real RHO,R,QMIT,EPRIM,EMAX,EMIN,H,E2,W,Q;
               array E1[-10:100];
               integer array COUNTS[-10:100];
               real procedure LANDAU(E,Q);
               real E,Q;
      begin    real LAMBDA,BB,R1,R2,R3,R4;
               AR:=0.154×Z×RHO×R/A;
               LAMBDA:=(Q-QMIT)/AR+ln(E/AR)-1.116;
               BB:=(0.00002×QMIT×(Z↑1.33333))/(AR×AR);
               R1:=BB+3.24;
               R2:=BB+4;
               R3:=BB+9;
               R4:=BB+25;
               LANDAU:=(0.3132/sqrt(R1))×exp(-(LAMBDA×LAMBDA)/R1)
                        +(0.116/sqrt(R2))×exp(-((LAMBDA-3)↑2)/R2)
                        +(0.057/sqrt(R3))×exp(-((LAMBDA-6.5)↑2)/R3)
                        +(0.035/sqrt(R4))×exp(-((LAMBDA-11)↑2)/R4)
        end    LANDAU;

               input statement for: Z,A,RHO,R,QMIT,EPRIM,EMAX,EMIN,H,N;
               for K:=1 step 1 until N do
               input statement for: E1[K],COUNTS[K];
               E1[0]:=2×E1[1]-E1[2];
               E1[N+1]:=2×E1[N]-E1[N-1];
               for E2:=EMAX step -H until EMIN do
      begin    W:=0;
               Q:=E1[1]-E2;
               for K:=1 step 1 until N do
```

```
begin  W:=(W+LANDAU(E1[K],Q)×(E1[K]/E1[1])×COUNTS[K]
          ×((E1[K-1]-E1[K])/2+(E1[K]-E1[K+1])/2));
       Q:=Q-(E1[K]-E1[K+1])
end;

       output statement for: E2,EPRIM-E2,W/H;
end    FALTUNG;
end
```

translated program

```
CLANDAUVERTEILUNG NACH BLUNCK-LEISEGANG KORRIGIERT
C(ZEITSCHRIFT FUER PHYSIK-130,641-649),
CMIT PRIMAERSPEKTRUM GEFALTET.

CHORST THEISSEN DEZ 1963


    DIMENSION E1(111)
    DIMENSION KOUNTS(111)
  2 FORMAT(2I5,7F10.4/I2)
    READ 2,NZ,NA,RHO,R,QMIT,EPRIM,EMAX,EMIN,H, N
  3 FORMAT(F6.3,6X,I5)
    READ 3,(E1(K+11),KOUNTS(K+11),K=1,N)
    E1(11)=2.*E1(12)-E1(13)
    E1(N+12)=2.*E1(N+11)-E1(N+10)
    N1=ABS((EMIN-EMAX)/(-H))+0.0000001 + 1.
    DO12 KI=1,NI
    AI=KI-1
    E2=AI*(-H)+EMAX
    W=0.0
    Q=E1(12)-E2
    NE=ABS((N-1)/1)+1
    DO 13 KE=1,NE
    K=(KE-1)*1+1
    COMMON NZ, NA, RHO, R, QMIT
    COUNTS=KOUNTS(K+11)
    W=(W+ANDAU(E1(K+11),Q)*(E1(K+11)/E1(12))*COUNTS
   1*((E1(K+10)-E1(K+11))/2.+(E1(K+11)-E1(K+12))/2.))
    Q=Q-(E1(K+11)-E1(K+12))
 13 CONTINUE
    D=EPRIM-E2
    F=W/H
 14 FORMAT(F7.3,10X,F6.3,10X,E11.4)
    PUNCH 14,E2,D,F
 12 CONTINUE
    END
    FUNCTION ANDAU(E,Q)
    COMMON NZ,NA,RHO,R,QMIT
    Z=NZ
    A=NA
    AR=0.154*Z*RHO*R/A
    AMBDA=(Q-QMIT)/AR+ALOG(E/AR)-1.116
    BB=(0.00002*QMIT*(Z**1.33333))/(AR*AR)
    R1=BB+3.24
    R2=BB+4.
    R3=BB+9.
    R4=BB+25.
    ANDAU=(0.3132/SQRT(R1))*EXP(-(AMBDA*AMBDA)/R1)
   1 +(0.116/SQRT(R2))*EXP(-((AMBDA-3.)**2)/R2)
   2 +(0.057/SQRT(R3))*EXP(-((AMBDA-6.5)**2)/R3)
   3 +(0.035/SQRT(R4))*EXP(-((AMBDA-11.)**2)/R4)
    RETURN
    END
```

6.1.5 A Simple Example Including Input Data and Computed Output Values

Remarks: This example is written in ALGOL 60 (except for the input/output statements) and translated into FORTRAN II. The input/output statements are written according to the rules of the NE computer (except that read and print appearing in the example are written with small letters). In order to run this ALGOL program, it was changed due to the hardware representation and restrictions of the NE (this modified program is not given here) and fed, together with the data given, into the NE 803B. The output values of this computation are presented.

The translated program, including the input data, was computed on a IBM 1620_{II}. The output values of this computation are also presented.

In order to compare the output values from the two programs, one must know that the numbers of the NE output are rounded and that the numbers of the IBM output are truncated; thus, a difference of ± 1 in the last digit is due to this procedure.

program

```
begin comment  LANDAUVERTEILUNG NACH BLUNCK-LEISEGANG KORRIGIERT
               (ZEITSCHRIFT FUER PHYSIK 130,641-649),
               MIT PRIMAERSPEKTRUM GEFALTET.

HORST THEISSEN DEZ 1963;

    integer N,Z,A,K;
    real RHO,R,QMIT,EPRIM,EMAX,EMIN,H,E2,W,Q;
    array E1[-10:100];
    integer array COUNTS[-10:100];
    real procedure LANDAU(E,Q);
    real E,Q;
    begin real LAMBDA,AR,BB,R1,R2,R3,R4;
        AR:=0.154×Z×RHO×R/A;
        LAMBDA:=(Q-QMIT)/AR+ln(E/AR)-1.116;
        BB:=(0.00002×QMIT×(Z↑1.33333))/(AR×AR);
        R1:=BB+3.24;
        R2:=BB+4;
        R3:=BB+9;
        R4:=BB+25;
        LANDAU:=(0.3132/sqrt(R1))×exp(-(LAMBDA×LAMBDA)/R1)
                +(0.116/sqrt(R2))×exp(-((LAMBDA-3)↑2)/R2)
                +(0.057/sqrt(R3))×exp(-((LAMBDA-6.5)↑2)/R3)
                +(0.035/sqrt(R4))×exp(-((LAMBDA-11)↑2)/R4)
    end;
    read Z,A,RHO,R,QMIT,EPRIM,EMAX,EMIN,H,N;
    for K:=1 step 1 until N do read E1[K],COUNTS[K];
```

```
print $$L10?
LANDAUVERTEILUNG EINSCHL. BLUNCK-LEISEGANG-KORREKTUR,
MIT DEM PRIMAERSPEKTRUM GEFALTET $L3??,
Z,$ = ORDNUNGSZAHL DES TARGETKERNES?,
A,$ = ATOMGEWICHT DES TARGETKERNES?,
RHO,$=DICHTE DES TARGETS (GR/CM-3)?,
R,$= DICKE DES TARGETS (CM)?
EPRIM,$= ENERGIE DER EINF ELEKTRONEN (MEV)?,
QMIT,$= MITTLERER ENERGIEVERLUST (MEV)$L3?
ENERGIE          ENERGIEVERLUST          LANDAU-
  (MEV)             (MEV)                VERTEILUNG      $L??;

    E1[0]:=2×E1[1]-E1[2];
    E1[N+1]:=2×E1[N]-E1[N-1];
    for E2:=EMAX step -H until EMIN do
    begin W:=0;
        Q:=E1[1]-E2;
        for K:=1 step 1 until N do
        begin W:=W+LANDAU(E1[K],Q)×(E1[K]/E1[1])×COUNTS[K]
                ×((E1[K-1]-E1[K])/2+(E1[K]-E1[K+1])/2);
            Q:=Q-(E1[K]-E1[K+1])

        end;

        print ALIGNED(2,3),SAMELINE,E2,$$S10??,EPRIM-E2,$$S10??
        SCALED(5),W/H;

    end
end
```

input data

```
6
12
2.0665
0.1545
0.599
53.382
53.2
52.3
0.05
23

53.509          227
53.490          474
53.470         1174
53.451         1931
53.431         3985
53.412         7084
53.393         7725
53.373         9309
53.354        10131
53,334        10084
53.315         9388
53.296         8199
53.276         6537
53.257         4836
53.237         3463
53.218         2452
53.198         1947
53.179         1389
53.140         1032
53.101          732
53.063          636
52.985          417
52.907          399
```

result output

```
LANDAUVERTEILUNG EINSCHL. BLUNCK-LEISEGANG-KORREKTUR
MIT DEM PRIMAERSPEKTRUM GEFALTET


     6 = ORDNUNGSZAHL DES TARGETKERNES
    12 = ATOMGEWICHT DES TARGETKERNES
2.0665000= DICHTE DES TARGETS (GR/CM-3)
.15450000= DICKE DES TARGETS (CM)
53.382000= ENERGIE DER EINF ELEKTRONEN (MEV)
.59900000= MITTLERER ENERGIEVERLUST (MEV)


ENERGIE      ENERGIEVERLUST      LANDAU-
 (MEV)            (MEV)         VERTEILUNG

 53.200           0.182          6.4727@-03
 53.150           0.232          1.2136@-00
 53.100           0.282          3.4180@+01
 53.050           0.332          2.7533@+02
 53.000           0.382          1.1167@+03
 52.950           0.432          2.3822@+03
 52.900           0.482          3.1976@+03
 52.850           0.532          3.0564@+03
 52.800           0.582          2.2981@+03
 52.750           0.632          1.5741@+03
 52.700           0.682          1.0749@+03
 52.650           0.732          7.8431@+02
 52.600           0.782          5.4104@+02
 52.550           0.832          4.1008@+02
 52.500           0.882          2.9780@+02
 52.450           0.932          2.2588@+02
 52.400           0.982          1.1222@+02
 52.350           1.032          6.0251@+01
 52.300           1.082          3.2770@+01
```

translated program

```
CLANDAUVERTEILUNG NACH BLUNCK-LEISEGANG KORRIGIERT
C(ZEITSCHRIFT FUER PHYSIK 130,641-649),
CMIT PRIMAERSPEKTRUM GEFALTET.
CHORST THEISSEN DEZ 1963
      DIMENSION E1(111)
      DIMENSION KOUNTS(111)
    2 FORMAT(2I5,6F10.4/F10.4,I2)
      READ 2,NZ,NA,RHO,R,QMIT,EPRIM,EMAX,EMIN,H, N
    3 FORMAT(F6.3,6X,I5)
      READ 3,(E1(K+11),KOUNTS(K+11),K=1,N)
    1 FORMAT(///////////53H LANDAUVERTEILUNG EINSCHL. BLUNCK-LEISEGANG-KO
     1 RREKTUR/33H MIT DEM PRIMAERSPEKTRUM GEFALTET///)
      PUNCH 1
    4 FORMAT(I9,32H = ORDNUNGSZAHL DES TARGETKERNES)
      PUNCH 4,NZ
    5 FORMAT(I9,31H = ATOMGEWICHT DES TARGETKERNES)
      PUNCH 5,NA
    6 FORMAT(F10.7,30H= DICHTE DES TARGETS (GR/CM-3))
      PUNCH 6,RHO
    7 FORMAT(F10.8,24H= DICKE DES TARGETS (CM))
      PUNCH 7,R
    8 FORMAT(F10.6,35H= ENERGIE DER EINF ELEKTRONEN (MEV))
      PUNCH 8,EPRIM
    9 FORMAT(F10.8,32H= MITTLERER ENERGIEVERLUST (MEV)///)
      PUNCH 9,QMIT
```

```
10  FORMAT(42HENERGIE          ENERGIEVERLUST          LANDAU-)
    PUNCH 10
11  FORMAT(45H  (MEV)                (MEV)             VERTEILUNG/)
    PUNCH 11
    E1(11)=2.*E1(12)-E1(13)
    E1(N+12)=2.*E1(N+11)-E1(N+10)
    NI=ABSF((EMIN-EMAX)/(-H))+0.0000001 + 1.
    DO12 KI=1,NI
    AI=KI-1
    E2=AI*(-H)+EMAX
    W=0.0
    Q=E1(12)-E2
    NE=ABSF((N-1)/1)+1
    DO 13 KE=1,NE
    K=(KE-1)*1+1
    COMMON NZ, NA, RHO, R, QMIT
    COUNTS=KOUNTS(K+11)
    W=(W+ANDAU(E1(K+11),Q)*(E1(K+11)/E1(12))*COUNTS
   1*((E1(K+10)-E1(K+11))/2.+(E1(K+11)-E1(K+12))/2.))
    Q=Q-(E1(K+11)-E1(K+12))
13  CONTINUE
    D=EPRIM-E2
    F=W/H
14  FORMAT(F7.3,10X,F7.3,10X,E11.4)
    PUNCH 14,E2,D,F
12  CONTINUE
    END

    FUNCTION ANDAU(E,Q)
    COMMON NZ,NA,RHO,R,QMIT
    Z=NZ
    A=NA
    AR=0.154*Z*RHO*R/A
    AMBDA=(Q-QMIT)/AR+LOGF(E/AR)-1.116
    BB=(0.00002*QMIT*(Z**1.33333))/(AR*AR)
    R1=BB+3.24
    R2=BB+4.
    R3=BB+9.
    R4=BB+25.
    ANDAU=(0.3132/SQRTF(R1))*EXPF(-(AMBDA*AMBDA)/R1)
   1 +(0.116/SQRTF(R2))*EXPF(-((AMBDA-3.)**2)/R2)
   2 +(0.057/SQRTF(R3))*EXPF(-((AMBDA-6.5)**2)/R3)
   3 +(0.035/SQRTF(R4))*EXPF(-((AMBDA-11.)**2)/R4)
    RETURN
    END
```

input data

```
   6   12        2.0665      .1545      .599      53.382      53.2      52.3
    .05   23
53.509             227
53.490             474
53.470            1174
53.451            1931
53.431            3985
53.412            7084
53.393            7725
53.373            9309
53.354           10131
53.334           10084
53.315            9388
53.296            8199
53.276            6537
53.257            4836
53.237            3463
53.218            2452
53.198            1947
53.179            1389
53.140            1032
53.101             732
53.063             636
52.985             417
52.907             399
```

output values

```
LANDAUVERTEILUNG EINSCHL. BLUNCK-LEISEGANG-KORREKTUR
MIT DEM PRIMAERSPEKTRUM GEFALTET


     6 = ORDNUNGSZAHL DES TARGETKERNES
    12 = ATOMGEWICHT DES TARGETKERNES
2.0665000= DICHTE DES TARGETS (GR/CM-3)
.15450000= DICKE DES TARGETS (CM)
53.382000= ENERGIE DER EINF ELEKTRONEN (MEV)
.59900000= MITTLERER ENERGIEVERLUST (MEV)
```

ENERGIE (MEV)	ENERGIEVERLUST (MEV)	LANDAU-VERTEILUNG
53.200	.182	6.4726E-03
53.150	.232	1.2137E-00
53.100	.282	3.4181E+01
53.050	.332	2.7533E+02
53.000	.382	1.1167E+03
52.950	.432	2.3822E+03
52.900	.482	3.1977E+03
52.850	.532	3.0564E+03
52.800	.582	2.2982E+03
52.750	.632	1.5740E+03
52.700	.682	1.0749E+03
52.650	.732	7.8430E+02
52.600	.782	5.4104E+02
52.550	.832	4.1008E+02
52.500	.882	2.9781E+02
52.450	.932	2.2588E+02
52.400	.982	1.1222E+02
52.350	1.032	6.0252E+01
52.300	1.082	3.2770E+01

6.1.6 A More Complex Example Including Input Data and Computed Output Values

Remarks: This example is written in ALGOL 60 (except for the input/output statements) and translated into FORTRAN II.

The input/output statements are written according to the rules of the NE computer (except that read and print appearing in the example are written with small letters). In order to run this ALGOL program, it was changed due to the hardware representation and restrictions of the NE (this modified program is not given here) and fed, together with the data given, into the NE 803B. The output values of this computation are presented.

The translated program, including the input data, was computed on a IBM 1620_{II}. The output values of this computation are also presented.

In order to compare the output values from the two programs, one must know that the numbers of the NE output are rounded and that the numbers of the IBM output are truncated; thus a difference of ± 1 in the last digit is due to this procedure.

program

```
begin LANDAUVERTEILUNG;
integer Z,A,K,KMAX,L,LMAX;
real QMIT,AR,BQU,TS,ENULL,T,R2,R1,R3,R4,NE,
    AA,BB,X1,X,C,ALPHA,MM,LE,QWRS,U,Q1,
    E,W,TY,GAMA,BETA,LAMBDA,CNORM,CMAX;
array E1,E2,C1,C2,Q,LA[-2:27];
real procedure GAMMA(X);
real X;
begin real XSTERN,G,X2,X3,X4;
      XSTERN:=X;
      if X < 10 then
      begin if X ≤ 0 then
            begin if X-entier(X)=0 then
                  begin print $GAMMA(?,SAMELINE,X,
                              $)= UNENDLICH?;
                        go to M
                  end
                  else XSTERN:=10+(entier(X)-X)
            end;
            if X > 0 then XSTERN:=10+(X-entier(X))
      end;
      X2:=XSTERN×XSTERN;
      X3:=X2×XSTERN;
      X4:=X3×XSTERN;
       G:=2.506628×exp(-XSTERN×(XSTERN-0.5)×ln(XSTERN))
            ×(1+0.08333333/XSTERN+0.0034722222/X2-0.00268132716/
            X3-0.000229472094/X4);
      if X < 10 then
      begin real N,F,F0;
            N:=1;
            F0:=9+(if X ≤ 0 then entier(X)-X
                  else X-entier(X));
            for F:=F0,F-1 while F ≥ entier(X) do
            N:=N×F;
            G:=G/N
```

```
      end;
      GAMMA:=G;
M: end GAMMA;

real procedure PHI(A,C,X);
value A,C,X;
real A,C,X;
begin real P,Q;
      integer J;
      Q:=P:=1;
      for J:=1,J+1 while abs(P) > 10-7 do
      begin P:=(P×(A+J)×X)/((C+J)×(1+J));
            Q:=Q+P
      end;
      PHI:=Q
end PHI;

real procedure PARCYLF(NY,Z);
value NY,Z;
real NY,Z;
begin if NY = 0 then PARCYLF:=exp(-.5×Z×Z);
      if NY ≠ 0 ∧ abs(Z) ⩾ 3 then
      begin integer J;
            real Q,P,MIN;
            if Z > 0 then
            begin Q:= 1;
            P:=-1;
            MIN:=10;
            for J:= 0,J+1 while abs(P) = MIN ∧ abs(P) > 10-5 do
               begin P:=-P×(NY-2×J)×(NY-2×J-1)/((J+1)×2×Z×Z);
                     Q:=Q+P;
                     if abs(P) < MIN then MIN:=abs(P)
               end;
            PARCYLF:=exp(-0.5×Z×Z)×(Z↑NY)×(Q-P)
            end;
            if Z < 0 then
            begin Z:=-Z;
            P:=1;
            Q:=1;
            MIN:=10;
            for J:=0,J+1 while abs(P) = MIN ∧ abs(P) > 10-5 do
               begin P:=P×(NY+2×J+1)×(NY+2×J+2)/(2×Z×Z×(J+1));
                     Q:=P+Q;
                     if abs(P) < MIN then MIN:=abs(P)
               end;
            PARCYLF:=2.506628×(Z↑(-NY-1))×(Q-P)/GAMMA(-NY)
            end
      end;
      if NY ≠ 0 ∧ abs(Z)     3 then
      begin real S1,S2;
            if (0.5-0.5×NY)-entier(0.5- 0.5×NY) = 0
            then S1:=0
            else S1:=1.772454×PHI(-0.5×NY,0.5,0.5×Z×Z)/GAMMA(0.5
                  -0.5×NY);
            if (-0.5×NY)-entier(-0.5×NY) = 0
            then S2:=0
            else S2:=3.544908×Z×PHI(0.5-0.5×NY,1.5,0.5×Z×Z)
                  /(1.414213×GAMMA(-0.5×NY));
            PARCYLF:=2↑(0.5×NY)×exp(-0.5×Z×Z)×(S1-S2)
      end
end PARCYLF;

real procedure LANDAU(E,Q1);
real E,Q1;
```

```
begin real LAMBDA,X1,X2,X3,X4,F,W;
      LAMBDA:=(Q1-QWRS)/AR-.05;
      X1:=1.4142×LAMBDA/sqrt(R1);
      X2:=1.4142×(LAMBDA-3)/sqrt(R2);
      X3:=1.4142×(LAMBDA-6.5)/sqrt(R3);
      X4:=1.4142×(LAMBDA-11)/sqrt(R4);
      F:=(.5↑(.5×TS))×(.3132×PARCYLF(-TS,-X1)/sqrt(R1↑NE)+.116×
          PARCYLF(-TS,-X2)/sqrt(R2↑NE)+.057×PARCYLF(-TS,-X3)/
          sqrt(R3↑NE)+.035×PARCYLF(-TS,-X4)/sqrt(R4↑NE));
      W:=((AR/ENULL)↑TS)×F;
      LANDAU:=W
end LANDAU;

PUSCHKIN:
      read Z,A,T,ENULL,AA,BB,X1,C,ALPHA,MM,CNORM,U;
      for K:=1 step 1 until 50 do
      begin read E1[K];
            if E1[K] = 4711 then begin KMAX:=K-1;
                                       go to VAT69
                                 end;
            E1[K]:=U×E1[K];
            read C1[K]
      end;

VAT69:
      GAMA:=ENULL/0.511;
      BETA:=sqrt(1-1/GAMA↑2);
      AR:=.1537×Z×T/(A×BETA×BETA);
      TS:=(T×.0013965333×Z×(Z+1)×ln(183/(Z↑.3333)))/
          (A×(1+.12×(Z/82)×(Z/82)));
      LE:=ln(sqrt)(GAMA×GAMA-1));
      X:=LE/2.3026;
      W:=1.022×sqrt(GAMA×GAMA-1)×BETA/21;
      TY:=(TS×TS)/(2×W×W);
      AA:=AA×T×(1+TY/TS)/(BETA×BETA);
      QMIT:=AA×(BB+.43+2×LE+ln(ENULL)-1
              -(if X ≤ X1 then 4.606×X+C+ALPHA×((X1-X)↑MM)
              else 4.606×X+C));
      BQU:=(.00002×QMIT×(Z↑1.3333))/(AR×AR);
      QWRS:=AA×(BB+1.06+2×LE+ln(AA)-1)-(if X ≤ X1 then
              4.606×X+C+ALPHA×((X1-X)↑MM) else 4.606×X+C)×AA
              +.05×AR×BQU+2×AR×TS;
      R1:=BQU+3.24;
      R2:=BQU+4;
      R3:=BQU+9;
      R4:=BQU+25;
      NE:=1-TS;

print $$L1o?
LANDAUVERTEILUNG EINSCHL BLUNCK-LEISEGANG-
UND BREMSSTRAHLUNGSKORREKTUR, MIT DEM
PRIMAERSPEKTRUM GEFALTET $L3??

Z,$ = ORDUNGSZAHL Z?,
A,$ = ATOMGEWICHT(A)?,
T,$= SCHICHTDICKE (G/CM**2)?,
$$L??,
AA/T,$= KONSTANTE GROSS A?,
BB,$= KONSTANTE GROSS B?,
X1,$= KONSTANTE X1?,
C,$= KONSTANTE C?,
ALPHA,$= KONSTANTE KLEIN A?,
MM,$= KONSTANTE KLEIN M?,
$$L??,
```

```
ENULL,$= ENERGIE DER EINF ELEKTRONEN (MEV)?,
QMIT,$= MITTLERER ENERGIEVERLUST (MEV)?,
QWRS,$= WAHRSCHEINLICHSTER ENERGIEVERLUST (MEV)?.
AR,$= A*R (MEV)?,
TS,$= SCHICHTDICKE/STRAHLUNGSLAENGE?,
BQU,$= B**2?,
$$L2?
FREQUENZ      ENERGIE      E-VERLUST   LAMBDA        VERTEILUNG
 ( MHZ)        (MEV)         (MEV)       ( 1)          (COUNTS)
$L??;
    L:=1;
    CMAX:=0;
    E1[0]:=2×E1[1]-E1[2];
    E1[KMAX+1]:=2×E1[KMAX]-E1[KMAX-1];
    for LAMBDA:=-5 step 1 until -1.9,-1.5 step
                  .5 until 3.1,4,5,6 step 2 until 16 do
    begin E2[L]:=ENULL--QWRS-AR×(LAMBDA+.05);
        C2[L]:=0;
        Q[L]:=E1[1]-E2[L];
        for K:=1 step 1 until KMAX do
        begin C2[L]:=C2[L]+LANDAU(E1[K],Q[L])×C1[K]
                    ((E1[K-1]-E1[K])/2
                    +(E1[K]-E1[K+1])/2);
            Q[L]:=Q[L]-(E1[K]-E1[K+1])
        end;
        C2[L]:=C2[L]/AR;
        LA[L]:=LAMBDA;
        if C2[L] > CMAX then CMAX:=C2[L];
        L:=L+1
    end;
    LMAX:=L-1;
    for L:=1 step 1 until LMAX do
    print SAMELINE,$S?,ALIGNED(2,3),E2[L]/U,PREFIX($$S5??),E2[L],
        ENULL-E2[L],LA[L],DIGITS(4),entier(C2[L]×CNORM+.5)
end LANDAUVERTEILUNG;
```

input data

```
6
12
4.98
54.130
0.0768
18.25
2
-3.22
.531
2.63
5824
3.88
14.010          363
13.995          551
13.985         1041
13.980         1757
13.975         3394
13.970         6185
13.965        10206
13.960        16135
13.955        20994
13.945        26488
13.930        25252
13.925        20946
```

```
13.920          16308
13.915          12407
13.910           9335
13.900           6803
13.890           5062
13.880           4161
13.870           3513
13.850           2728
13.810           1696
13.770           1125
13.730            968
4711
```

result output

```
LANDAUVERTEILUNG EINSCHL BLUNCK-LEISEGANG-
UND BREMSSTRAHLUNGSKORREKTUR, MIT DEM
PRIMAERSPEKTRUM GEFALTET


      6 = ORDNUNGSZAHL Z
     12 = ATOMGEWICHT(A)
 4.9800000= SCHICHTDICKE(G/CM**2)

 .07696900= KONSTANTE GROSS A
 18.250000= KONSTANTE GROSS B
 2.0000000= KONSTANTE X1
-3.2200000= KONSTANTE C
 .53100000= KONSTANTE KLEIN A
 2.6300000= KONSTANTE KLEIN M

 54.130000= ENERGIE DER EINF ELEKTRONEN (MEV)
 9.5403885= MITTLERER ENERGIEVERLUST (MEV)
 7.9705478= WAHRSCHEINLICHSTER ENERGIEVERLUST (MEV)
 .38274711= A*R (MEV)
 .11219839= SCHICHTDICKE/STRAHLUNGSLAENGE
 .01419977= B**2

 FREQUENZ  ENERGIE    E-VERLUST    LAMBDA     VERTEILUNG
  (MHZ)     (MEV)      (MEV)       (1)        (COUNTS)

   12.385    48.054     6.076     -5.000          1
   12.286    47.671     6.459     -4.000         23
   12.188    47.289     6.841     -3.000        211
   12.089    46.906     7.224     -2.000       1084
   12.040    46.714     7.416     -1.500       1982
   11.990    46.523     7.607     -1.000       3153
   11.941    46.332     7.798     -0.500       4381
   11.892    46.140     7.990      0.000       5355
   11.843    45.949     8.181      0.500       5824
   11.793    45.758     8.372      1.000       5740
   11.744    45.566     8.564      1.500       5271
   11.695    45.375     8.755      2.000       4670
   11.645    45.183     8.947      2.500       4130
   11.596    44.992     9.138      3.000       3702
   11.497    44.609     9.521      4.000       3018
   11.399    44.227     9.903      5.000       2361
   11.300    43.844    10.286      6.000       1850
   11.103    43.078    11.052      8.000       1361
   10.905    42.313    11.817     10.000       1005
   10.708    41.547    12.583     12.000        754
   10.511    40.782    13.348     14.000        598
   10.313    40.016    14.114     16.000        469
```

translated progam

```
C LANDAUVERTEILUNG
      DIMENSION E1(30),E2(30),C1(30),C2(30),Q(30),ALA(30)
      COMMON E1,E2,C1,C2,ALA,L,KMAX,CMAX,QWRS,AR,ENULL,R1,R2,R3,R4,TS,
     1 ANE
 1001 FORMAT(I2)
  101 READ 1001,NZ
      Z=NZ
 1002 FORMAT(I3)
      READ 1002,NA
      A=NA
 1003 FORMAT(4F10.4)
      READ 1003,T,ENULL,AA,BB
      READ 1001,NX1
      X1=NX1
      READ 1003,C,ALPHA,AMM,CNORM
 1004 FORMAT(F5.2)
      READ 1004,U
 1005 FORMAT(F9.3)
      DO 200 K=1,100
      READ 1005,E1(K+3)
      IF(E1(K+3)-4711.)201,202,201
  202 KMAX=K-1
      GO TO 102
  201 CONTINUE
      E1(K+3)=U*E1(K+3)
 1006 FORMAT(F8.1)
      READ 1006,C1(K+3)
  200 CONTINUE
  102 GAMA=ENULL/0.511
      BETA=SQRTF(1.-1./GAMA**2)
      AR=0.1537*Z*T/(A*BETA*BETA)
      TS=(T*0.0013965333*Z*(Z+1.)*LOGF(183./(Z**0.3333)))/
     1 (A*(1.+0.12*(Z/82.)*(Z/82.)))
      ALE=LOGF(SQRTF(GAMA*GAMA-1.))
      X=ALE/2.3026
      W=1.022*SQRTF(GAMA*GAMA-1.)*BETA/21.
      TY=(TS*TS)/(2.*W*W)
      AA=AA*T*(1.+TY/TS)/(BETA*BETA)
      IF(X-X1)300,300,301
  300 QMIT=AA*(BB+.43+2.*ALE+LOGF(ENULL)-1.-(4.606*X+C+ALPHA*((X1-X)**MM
     1 )))
      GO TO 302
  301 QMIT=AA*(BB+.43+2.*ALE+LOGF(ENULL)-1.-(4.606*X+C))
  302 CONTINUE
      BQU=(0.00002*QMIT*(Z**1.3333))/(AR*AR)
      IF(X-X1)303,303,304
  303 QWRS=AA*(BB+1.06+2.*ALE+LOGF(AA)-1.)-(4.606*X+C+ALPHA*((X1-X)**MM)
     1 )*AA+.05*AR*BQU+2.*AR*TS
      GO TO 305
  304 QWRS=AA*(BB+1.06+2.*ALE+LOGF(AA)-1.)-(4.606*X+C)*AA+.05*AR*BQU+2.*
     1 AR*TS
  305 CONTINUE
      R1=BQU+3.24
      R2=BQU+4.
      R3=BQU+9.
      R4=BQU+25.
      ANE=1.-TS
  400 FORMAT(//////////42HLANDAUVERTEILUNG EINSCHL BLUNCK-LEISEGANG-/37H
     1 UND BREMSSTRAHLUNGSKORREKTUR, MIT DEM/
     2 24HPRIMAERSPEKTRUM GEFALTET///)
      PUNCH 400
```

```
  401  FORMAT(/I9,17H = ORDNUNGSZAHL Z/I9,17H = ATOMGEWICHT(A)/
     1 F10.7,24H= SCHICHTDICKE (G/CM**2))
       PUNCH 401,Z,A,T
  402  FORMAT(/F10.8,19H= KONSTANTE GROSS A/F10.6,19H= KONSTANTE GROSS B/
     1 F10.7,14H= KONSTANTE X1/F10.7,13H= KONSTANTE C/F10.8,19H= KONSTANT
     2 E KLEIN A/F10.7,19H= KONSTANTE KLEIN M)
       TAA=AA/T
       PUNCH 402,TAA,BB,X1,C,ALPHA,AMM
  403  FORMAT(/F10.6,35H= ENERGIE DER EINF ELEKTRONEN (MEV)/
     1 F10.7,32H= MITTLERER ENERGIEVERLUST (MEV)/
     2 F10.7,41H= WAHRSCHEINLICHSTER ENERGIEVERLUST (MEV)/F10.8,11H= A*R
     3 (MEV)/F10.8,31H= SCHICHTDICKE/STRAHLUNGSLAENGE/F10.8,6H= B**2)
       PUNCH 403,ENULL,QMIT,QWRS,AR,TS,BQU
  404  FORMAT(//61HFREQUENZ        ENERGIE       E-VERLUST     LAMBDA       VER
     1 TEILUNG/60H ( MHZ)          (MEV)          (MEV)        (1)          (CO
     2 UNTS)//)
       PUNCH 404
       L=1
       CMAX=0.
       E1(3)=2.*E1(4)-E1(5)
       E1(KMAX+4)=2.*E1(KMAX+3)-E1(KMAX+2)
       NI1=ABSF(-1.9+5.)+1.00000000001
       DO 500 NA1=1,NI1
       AI1=NA1-1
       AMBDA=AI1-5.
       CALL MESS(AMBDA)
  500  CONTINUE
       NI2=ABSF((3.1+1.5)/0.5)+1.00000000001
       DO 503 NA2=1,NI2
       AI2=NA2-1
       AMBDA=AI2*0.5-1.5
       CALL MESS(AMBDA)
  503  CONTINUE
       AMBDA=4.0
       CALL MESS(AMBDA)
       AMBDA=5.0
       CALL MESS(AMBDA)
  507  NI3=ABSF((16.-6.)/2.)+1.00000000001
       DO 509 NA3=1,NI3
       AI3=NA3-1
       AMBDA=AI3*2.+6.
       CALL MESS(AMBDA)
  509  CONTINUE
       LMAX=L-1
       DO 510 L=1,LMAX
       IF(C2(L+3)*CNORM/CMAX+0.5)511,512,512
  511  NUT=C2(L+3)*CNORM/CMAX+0.5-1.
       GO TO 513
  512  NUT=C2(L+3)*CNORM/CMAX+0.5
  513  CONTINUE
       DOG=E2(L+3)/U
       HOUND=E2(L+3)
       CAT=ENULL-E2(L+3)
       TIGER=ALA(L+3)
  514  FORMAT(1X,F7.3,5X,F7.3,5X,F7.3,5X,F7.3,5X,I5)
       PUNCH 514,DOG,HOUND,CAT,TIGER,NUT
  510  CONTINUE
       END
C      LANDAUVERTEILUNG
```

```
      SUBROUTINE MESS(AMBDA)
      DIMENSION E1(30),E2(30),C1(30),C2(30),Q(30),ALA(30)
      COMMON E1,E2,C1,C2,ALA,L,KMAX,CMAX,QWRS,AR,ENULL,R1,R2,R3,R4,TS,
     1 ANE
  501 E2(L+3)=ENULL-QWRS-AR*(AMBDA+0.05)
      C2(L+3)=0.
      Q(L+3)=E1(4)-E2(L+3)
      DO 601 K=1,KMAX
      C2(L+3)=C2(L+3)+ANDAU(E1(K+3),Q(L+3))*C1(K+3)*
     1 ((E1(K+2)-E1(K+3))/2.
     2 +(E1(K+3)-E1(K+4))/2.)
      Q(L+3)=Q(L+3)-(E1(K+3)-E1(K+4))
  601 CONTINUE
      C2(L+3)=C2(L+3)/AR
      ALA(L+3)=AMBDA
      IF(C2(L+3)-CMAX)602,602,603
  603 CMAX=C2(L+3)
  602 CONTINUE
      L=L+1
      RETURN
      END


 FUNCTION ANDAU(E,Q1)
 DIMENSION E1(30),E2(30),C1(30),C2(30),Q(30),ALA(30)
      COMMON E1,E2,C1,C2,ALA,L,KMAX,CMAX,QWRS,AR,ENULL,R1,R2,R3,R4,TS,
     1 ANE
      AMBDA=(Q1-QWRS)/AR-0.05
      X1=1.4142*AMBDA/SQRTF(R1)
      X2=1.4142*(AMBDA-3.)/SQRTF(R2)
      X3=1.4142*(AMBDA-6.5)/SQRTF(R3)
      X4=1.4142*(AMBDA-11.)/SQRTF(R4)
      F=(.5**(.5*TS))*
     1 (.3132*PARCYL(-TS,-X1)/SQRTF(R1**ANE)
     2 +.116*PARCYL(-TS,-X2)/SQRTF(R2**ANE)
     3 +.057*PARCYL(-TS,-X3)/SQRTF(R3**ANE)
     4 +.035*PARCYL(-TS,-X4)/SQRTF(R4**ANE))
      W=((AR/ENULL)**TS)*F
      ANDAU=W
      RETURN
C ANDAU
      END


 FUNCTION PARCYL(Y,Z)
 IF(Y)10,11,10
   11 PARCYL=EXPF(-0.5*Z*Z)
   10 CONTINUE
      IF(Y)12,13,12
   12 IF(ABSF(Z)-3.)13,13,15
   15 IF(Z)16,16,17
   17 Q=1.
      P=-1.
      AMIN=10.
      J=0
   24 AJ=J
      P=-P*(Y-2.*AJ)*(Y-2.*AJ-1.)/((AJ+1.)*2.*Z*Z)
      Q=Q+P
      IF(ABSF(P)-AMIN)25,26,26
   25 AMIN=ABSF(P)
   26 CONTINUE
   20 J=J+1
      IF(ABSF(P)-AMIN)27,28,27
   28 IF(ABSF(P)-1.0E-05)27,27,30
```

```
30  AJ=J
    P=-P*(Y-2.*AJ)*(Y-2.*AJ-1.)/((AJ+1.)*2.*Z*Z)
    Q=Q+P
    IF(ABSF(P)-AMIN)31,32,32
31  AMIN=ABSF(P)
32  CONTINUE
    GO TO 20
27  CONTINUE
    PARCYL=EXPF(-0.5*Z*Z)*(Z**Y)*(Q-P)
16  CONTINUE
    IF(Z)33,34,34
33  Z=-Z
    P=1.
    Q=1.
    AMIN=10.
36  J=0
    AJ=J
    P=P*(Y+2.*AJ+1.)*(Y+2.*AJ+2.)/(2.*Z*Z*(AJ+1.))
    Q = P+Q
    IF(ABSF(P)-AMIN)42,43,43
42  AMIN=ABSF(P)
43  CONTINUE
35  J=J+1
    IF(ABSF(P)-AMIN)44,45,44
45  IF(ABSF(P)-1.0E-05)44,44,47
47  AJ=J
    P=P*(Y+2.*AJ+1.)*(Y+2.*AJ+2/(2.*Z*Z*(AJ+1.))
    Q=P+Q
    IF(ABSF(P)-AMIN)48,49,49
48  AMIN=ABSF(P)
49  CONTINUE
    GO TO 35
44  CONTINUE
    PARCYL=2.506628*(Z**(-Y-1.))*(Q-P)/AMMA(-Y)
34  CONTINUE
13  CONTINUE
    IF(Y)50,51,50
50  IF(ABSF(Z)-3.)52,52,51
52  IF(0.5—0.5*Y)54,55,55
54  NP=0.5-0.5*Y-1.
    GO TO 56
55  NP=0.5-0.5*Y
56  CONTINUE
    P=NP
    IF((0.5-0.5*Y)-P)57,58,57
58  S1=0.
    GO TO 59
57  AB=-0.5*Y
    XB=0.5*Z*Z
    XC=0.5-0.5*Y
    S1=1.772454*PHI(AB,0.5,XB)/AMMA(XC)
59  CONTINUE
    IF(-0.5*Y)60,61,61
60  NR=-0.5*Y-1.
    GO TO 62
61  NR=-0.5*Y
62  CONTINUE
    R=NR
    IF(-0.5*Y-R)63,64,63
64  S2=0.
    GO TO 65
63  DOOF=0.5-0.5*Y
```

```
      DUMM=0.5*Z*Z
      DEP=-0.5*Y
      S2=3.544908*Z*PHI(DOOF,1.5,DUMM)/(1.414213*AMMA(DEP))
   65 CONTINUE
      PARCYL=2.**(0.5*Y)*EXPF(-0.5*Z*Z)*(S1-S2)
   51 CONTINUE
      RETURN
C PARCYL
      END

      FUNCTION AMMA(X)
      XSTERN=X
      IF(X-10.)10,11,11
   10 IF(X)12,12,13
   12 IF(X)14,15,15
   14 NX=X-1.
      GO TO 16
   15 NX=X
   16 CONTINUE
      PX=NX
      IF(X-PX)17,81,17
   18 FORMAT(5HAMMA(,E18.12,12H)=  UNENDLICH)
   81 PUNCH 18,X
      GO TO 103
   17 XSTERN=10.+(PX-X)
   13 CONTINUE
      IF(X)11,11,19
   19 XSTERN=10.+(X-PX)
   11 CONTINUE
      X2=XSTERN*XSTERN
      X3=X2*XSTERN
      X4=X3*XSTERN
      G=2.506628*EXPF(-XSTERN+(XSTERN-0.5)*LOGF(XSTERN))
     1*(1.+0.08333333/XSTERN+0.0034722222/X2-0.00268132716/
     2 X3-0.000229472094/X4)
      IF(X-10.)20,21,21
   20 AN=1.
      IF(X)8,9,9
    8 NX=X-1.
      GO TO 7
    9 NX=X
    7 CONTINUE
      PX=NX
      IF(X)22,22,23
   22 F0=9.+PX-X
      GO TO 24
   23 F0=9.+X-PX
   24 CONTINUE
      F=F0
      AN=AN*F
   33 F=F-1.
      IF(F-PX)30,31,31
   31 AN=AN*F
      GO TO 33
   30 CONTINUE
   36 G=G/AN
   21 CONTINUE
      AMMA=G
  103 RETURN
C AMMA
      END
```

```
      FUNCTION PHI(A,C,X)
      Q=1.
           P=1.
           J=0
           AJ=J
           P=(P*(A+AJ)*X)/((C+AJ*(1.+AJ))
           Q=Q+P
        13 J=J+1
           IF(ABSF(P)-1.0E-07)15,15,16
        16 AJ=J
           P=(P*(A+AJ)*X)/((C+AJ)*(1.+AJ))
           Q=Q+P
           GO TO 13
        15 CONTINUE
           PHI=Q
           RETURN
C PHI
           END
```

input data

```
 6
 12
     4.98          54.13          0.0768   18.25
 2
   -3.22            .531          2.63    5824.
  3.88
    14.010
    363.
    13.995
    551.
    13.985
   1041.
    13.980
   1757.
    13.975
   3394.
    13.970
   6185.
    13.965
  10206.
    13.960
  16135.
    13.955
  20994.
    13.945
  26488.
    13.930
  25252.
    13.925
  20946.
    13.920
  16308.
    13.915
  12407.
    13.910
   9335.
    13.900
   6803.
    13.890
   5062.
    13.880
```

```
 4161.
  13.870
 3513.
  13.850
 2728.
  13.810
 1696.
  13.770
 1125.
  13.730
  968.
4711.
```

output values

```
LANDAUVERTEILUNG EINSCHL BLUNCK-LEISEGANG-
UND BREMSSTRAHLUNGSKORREKTUR, MIT DEM
PRIMAERSPEKTRUM GEFALTET


     6= ORDNUNGSZAHL Z
    12= ATOMGEWICHT(A)
4.9800000= SCHICHTDICKE (G/CM**2)

.07696899= KONSTANTE GROSS A
18.250000= KONSTANTE GROSS B
2.0000000= KONSTANTE X1
-3.2200000= KONSTANTE C
.53100000= KONSTANTE KLEIN A
2.6300000= KONSTANTE KLEIN M

54.130000= ENERGIE DER EINF ELEKTRONEN (MEV)
9.5403880= MITTLERER ENERGIEVERLUST (MEV)
7.9705479= WAHRSCHEINLICHSTER ENERGIEVERLUST (MEV)
.38274712= A*R (MEV)
.11219838= SCHICHTDICKE/STRAHLUNGSLAENGE
.01419976= B**2
```

FREQUENZ (MHZ)	ENERGIE (MEV)	E-VERLUST (MEV)	LAMBDA (1)	VERTEILUNG (COUNTS)
12.385	48.054	6.075	−5.000	1
12.286	47.671	6.458	−4.000	23
12.187	47.288	6.841	−3.000	211
12.089	46.905	7.224	−2.000	1084
12.039	46.714	7.415	−1.500	1982
11.990	46.523	7.606	−1.000	3153
11.941	46.331	7.798	−.500	4381
11.891	46.140	7.989	0.000	5355
11.842	45.948	8.181	.500	5824
11.793	45.757	8.372	1.000	5740
11.743	45.566	8.563	1.500	5271
11.694	45.374	8.755	2.000	4670
11.645	45.183	8.946	2.500	4130
11.595	44.992	9.137	3.000	3706
11.497	44.609	9.520	4.000	3018
11.398	44.226	9.903	5.000	2361
11.299	43.843	10.286	6.000	1850
11.102	43.078	11.051	8.000	1361
10.905	42.312	11.817	10.000	1005
10,708	41.547	12.582	12.000	754
10.510	40.781	13.348	14.000	598
10.313	40.016	14.113	16.000	469

6.2 FORTRAN INTO ALGOL 60

On the following pages the method of translating a FORTRAN program into an ALGOL 60 program is described. The translation of the program itself is explained, together with a comment on how to adapt the input data. The input/output statements are not treated here, they are discussed in Part 3.

Two different FORTRAN II programs (including data input and computed results) and their translations into ALGOL* serve as examples to illustrate the method of translation.

The FORTRAN II programs, were computed on the IBM 1620_{II} (40,000 character memory, punched cards input/output) and the ALGOL programs were run on the NE 803B (8K memory, paper tape input/output).†

6.2.1 Translation from a FORTRAN II Program

To translate a FORTRAN II program into an ALGOL program, it is recommended that the following patterns be followed:

1. Scan the main program and the *COMMON* statements of the subprograms for integer variable identifiers. The first character of these integer variable identifiers is either *I*, *J*, *K*, *L*, *M*, or *N*. Declare such identifiers in the beginning of the ALGOL program under **integer**.

2. Scan the main program and the *COMMON* statements of the subprograms for real variable identifiers. The first character of these real variable identifiers is any alphabetic character except *I*, *J*, *K*, *L*, *M*, or *N*. Declare such identifiers in the beginning of the program under **real**.

3. Look for *GO TO*(. . .),*Nv* statements in the main program, because their translation (see Part 5 "*GO TO*") leads to a switch declaration in the beginning of the program.

4. Look for arithmetic statements, functions, and subprograms, their translation must be placed before the translation of the main program.

5. If one reaches a format statement or an input/output statement, then there are two possibilities:

(*a*) If the input/output of the ALGOL processor follows the FORTRAN rules (see Part 3), then one must copy the format statements and the input/output statements.

(*b*) If the input/output of the ALGOL processor follows other rules than FORTRAN the format statements must be bypassed. They must

* An ALGOL program is an ALGOL 60 program which includes input/output statements.

† The program itself contains only the NE input/output statements, the ALGOL 60 reference language symbols are not altered due to the hardware representation of the NE.

be translated together with the appropriate input/output statements with the aid of Part 3.

Note: Most ALGOL processors do not need any specification on the input data (such as format statements). Most of the ALGOL processors allow the print-out of the desired numbers (standard format) with ϵ (or δ) decimal digits times a power of ten if no declaration of desired format is made.

6. Before translating line by line (or sometimes group of statements by group of statements) with the aid of Part 5, it is useful to keep the following points in mind:

(i) Numbers terminating with a decimal point can be written without the decimal point.

(ii) Do not forget to place a semicolon after every statement.

(iii) A semicolon before **end** is not necessary.

(iv) After translating the program, check whether for every **begin** here is an appropriate **end**.

(v) After translation of the program, the sequence of the statements must be as followes. (If this is not the case, the sequence must be changed in the proper way.):

integer
real
array and/or **integer array**
switch
subprograms
main program

The sequence in a subprogram (i.e. **procedure** and/or **real procedure** and/or **integer procedure**) is:

(*a*) type of procedure (**real procedure**, **integer procedure**, **procedure**) and subprogram identifier with (if any) the appropriate parameters

(*b*) **value**

(*c*) specifications (**integer**, **real**, **label**, **integer array**, **array**, **switch**, **string**, **procedure**, **real procedure**, **integer procedure**)

(*d*) subprogram.

Comment records may be placed anywhere.

6.2.2 Translation from a FORTRAN IV Program

To translate a FORTRAN IV program into an ALGOL 60 program, it is recommended that the following pattern be followed:

1. Scan the main program and the *COMMON* statements of the subprograms for integer variable identifiers which are not declared under *INTEGER* or *REAL* or *LOGICAL* in the FIV program (N30). The

first character of these integer variable identifiers is either *I*, *J*, *K*, *L*, *M*, or *N*. If any of these identifiers are found, declare them in the ALGOL program under **integer**, together with the identifiers listed under *INTEGER* in the FIV program.

2. Scan the main program and the *COMMON* statements of the subprograms for real variable identifiers which are not declared under *REAL* or *INTEGER* or *LOGICAL* in the FIV program (N30). The first character of these real variable identifiers is any alphabetic character except *I*, *J*, *K*, *L*, *M*, or *N*. If any of these identifiers are found, declare them in the ALGOL program under **real**, together with the identifiers listed under *REAL* in the FIV program.

3. Look for *GO TO*(. . .),*Nv* statements in the main program, because their translation (see Part 5 "*GO TO*") leads to a switch declaration in the beginning of the program.

4. Look for arithmetic statement functions and subprograms, their translation must be placed before the translation of the main program.

5. If one reaches a format statement or an input/output statement, then there are two possibilities:

(*a*) If the input/output of the ALGOL processor follows the FORTRAN rules (see Part 3), then one must copy the format statements and the input/output statements.

(*b*) If the input/output of the ALGOL processor follows other rules than FORTRAN the format statements must be bypassed. They must be translated together with the appropriate input/output statements with the aid of Part 3.

Note: Most ALGOL processors do not need any specification on the input data (such as format statements). Most of the ALGOL processors allow the print-out of the desired numbers (standard format) with ϵ (or δ) decimal digits times a power of ten if no declaration of desired format is made.

6. Before translating line by line (or sometimes group of statements by group of statements) with the aid of Part 5, it is useful to keep the following points in mind:

(i) Numbers terminating with a decimal point can be written without the decimal point.

(ii) Do not forget to place a semicolon after every statement.

(iii) A semicolon before **end** is not necessary.

(iv) After translating the program, check whether for every **begin** there is an appropriate **end**.

(v) After translation of the program, the sequence of the statements must be as follows. (If this is not the case the sequence must be changed in the proper way.)

integer
real
Boolean
array and/or **integer array**
switch
subprograms
main program

The sequence in a subprogram (i.e. **procedure** and/or **real procedure** and/or **integer procedure** and/or **Boolean procedure**) is:

(*a*) type of subprogram (**real procedure**, **integer procedure**, **Boolean procedure**, **procedure**) and subprogram identifier with (if any) the appropriate parameters

(*b*) **value**

(*c*) specifications (**integer**, **real**, **Boolean**, **label**, **integer array**, **array**, **switch**, **string**, **procedure**, **real procedure**, **integer procedure**, **Boolean procedure**)

(*d*) subprogram.

Comment records may be placed anywhere.

6.2.3 The Data

To translate input data of a FORTRAN program into input data of an ALGOL program, one should keep the following points in mind:

1. If the ALGOL processor follows the FORTRAN input rules (see Section 3) the input data may be copied directly.

2. If the ALGOL processor does not follow the FORTRAN input rules, then the different input numbers or values must be separated. (This is opposite to FORTRAN, where a separation of the numbers or values is not necessary because the numbers or values are identified with the aid of the appropriate format statements.) Although ALGOL processors allow any character as a number or value separator except 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, −, +, ., or $_{10}$, it is recommended to separate by using a new record.

3. Blank spaces on an input data record in FORTRAN are evaluated as zeros.

4. After keeping these points in mind, translate the input data with the aid of Part 5 "numbers".

6.2.4 A Simple Example Including Input Data and Computed Output Values

Remarks: This example is written in FORTRAN II and translated into ALGOL.

The program, including the input data given, was run on a IBM 1620_{II}. The output values of this computation are presented. The

input/output statements are translated according to the rules of the NE computer (except that read and print appearing in the example are written with small letters). In order to run the translated program, it was changed due to the hardware representation and restrictions of the NE (this modified program is not given here) and fed, together with the data given, into the NE 803B. The output values of this computation are also given.

In order to compare the output values from the two programs, one must know that the numbers of the NE output are rounded and that the numbers of the IBM output are truncated; thus a difference of ± 1 in the last digit is due to this procedure.

program

```
C   POLYNOMIAL CURVE FITTING
        DIMENSION A(12,13)
    2   JV=13
        READ 3,XMN,XRA,MIND,INC,MAXD
    3   FORMAT(2F10.3,3I5)
        ARX=1.0/XRA
        IF(MIND)4,6,6
    4   PRINT 5
    5   FORMAT(8HMIND NEG)
        GO TO 69
    6   MAXI=MAXD+1
        IF(JV-MAXI)7,7,8
    7   MAXI=JV-1
    8   KMIN=MIND+1
C   CLEAR TOTAL BOXES
        A(1,JV)=0.0
        YSQ=0.0
        NP=0
        DO 9 J=2,MAXI
        A(J,JV)=0.0
        M=J-1
        DO 9 I=M,J
    9   A(I,J)=0.0
C   READ DATA AND ACCUMULATE SUMS OF POWERS
   10   READ 11,P,Q,J
   11   FORMAT(F10.4,F10.4,I1)
        IF(J)16,12,16
   12   NP=NP+1
        S=(P-XMN)*ARX
        YSQ=YSQ+Q*Q
        T=1.0
        K=1
        DO 15 J=2,MAXI
        M=J-1
        DO 15 I=M,J
        IF(K-MAXI)13,13,14
   13   A(K,JV)=A(K,JV)+T*Q
        K=K+1
   14   T=T*S
   15   A(I,J)=A(I,J)+T
        GO TO 10
   16   PUNCH 3,XMN,XRA,MIND,INC
        D=NP
        A(1,1)=D
        N=(MAXI-KMIN)/INC
        IF(NP-N)17,17,18
```

```
   17  N=NP-1
   18  MAXD=N*INC+MIND
       MAXI=MAXD+1
C  SPREAD TOTALS THROUGHOUT TOP HALF OF MATRIX
       M1=MAXI-2
       DO 20 I=KMIN,M1,INC
       DO 20 J=I,MAXI,INC
       IF(J-I-2)20,19,19
   19  L=I+J
       K=L/2
       L=L-K
       A(I,J)=A(K,L)
   20  CONTINUE
       PUNCH 21,MAXD,NP,YSQ
   21  FORMAT(7HMAX DEG I3,11H NO POINTS I5,12H RAW S OF S E11.4)
C  SET UP TO SOLVE EQUATIONS
       DO 42 K=KMIN,MAXI,INC
       P=1.0/A(K,K)
       A(K,K)=(K,JV)*P
C  SOLVE FOR RESIDUALS
       YSQ=YSQ-A(K,K)*A(K,JV)
       D=D-1.0
       IF(D)22,22,23
   22  Q=0.9999E99
       GO TO 26
   23  IF(YSQ)24,24,25
   24  Q=0.0
       GO TO 26
   25  Q=SQRTF(YSQ/D)
   26  J=K-1
       PUNCH 27,J,YSQ,Q
   27  FORMAT(8HPOLY DEG I3,13H RESID S OF S E11.4,7H ST ERR E11.4)
C  SOLVE FOR COEFFICIENTS
       M1=K+INC
       M=MAXI-M1
       IF(M)30,28,28
   28  DO 29 J=M1,MAXI,INC
       A(J,K)=A(K,J)
   29  A(K,J)=A(K,J)*P
   30  L=1
       N=KMIN
       T=1.0
       DP 41 I=1,MAXI
       IF(N-I)4,31,40
   31  N=N+INC
       IF(I-K)33,37,32
   32  M1=I
   33  P=A(I,K)
       IF(M)36,34,34
   34  DO 35 J=M1,MAXI,INC
   35  A(I,J)=A(I,J)-A(K,J)*P
   36  A(I,K)=A(I,JV)-A(K,K)*P
       GO TO(38,41),L
   37  L=2
   38  J=I-1
       R=A(I,K)*T
       PUNCH 39,J,R
   39  FORMAT(I4,10H DEG COEF E11.4)
   40  T=T*ARX
   41  CONTINUE
   42  JV=K
   69  CONTINUE
       END
```

input data

```
  0.0          2.0          0       1       3
 -0.9          0.500  0
 -0.7          0.977  0
 -0.5          1.199  0
 -0.3          1.345  0
 -0.1          1.454  0
  0.1          1.541  0
  0.3          1.614  0
  0.5          1.676  0
  0.7          1.730  0
  0.9          1.779  0
  0.0          0.0    1
```

output values

```
 0.000E-99 2.000E-00          0       1
MAX DEG   3 NO POINTS    10 RAW S OF S  2.0511E+01
POLY DEG  0 RESID S OF S 1.4262E-00 ST ERR 3.9808E-01
    0 DEG COEF   1.3815E-00
POLY DEG  1 RESID S OF S 2.0671E-01 ST ERR 1.6074E-01
    0 DEG COEF   1.3815E-00
    1 DEG COEF   6.0790E-01
POLY DEG  2 RESID S OF S 4.3332E-02 ST ERR 7.8678E-02
    0 DEG COEF   1.5266E-00
    1 DEG COEF   6.0790E-01
    2 DEG COEF  -4.3977E-01
POLY DEG  3 RESID S OF S 9.2597E-03 ST ERR 3.9284E-02
    0 DEG COEF   1.5266E-00
    1 DEG COEF   3.6462E-01
    2 DEG COEF  -4.3977E-01
    3 DEG COEF   4.1516E-01
```

translated program

```
begin comment POLYNOMIAL CURVE FITTING;
     integer JV,MIND,MAXD,MAXI,KMIN,NP,J,M,I,K,N,M1,L;
     real XMN,XRA,ARX,YSQ,P,Q,S,T,D,R;
     array A[1:12,1:13];
     switch JUMP:=38,41;
           JV:=13;
           read XMN,XRA,MIND,INC,MAXD;
           ARX:=1/XRA;
           if MIND ≥ 0 then go to 6 else go to 4;
      4:   print$MIND NEG?;
           go to 69;
      6:   MAXI:=MAXD+1;
           if (JV-MAXI) ≤ 0 then go to 7 else go to 8;
      7:   MAXI:=JV-1;
      8:   KMIN:=MIND+1;
     comment CLEAR TOTAL BOXES;
           A[1,JV]:=0;
           YSQ:=0;
           NP:=0;
           for J:=1 step 1 until MAXI do
           begin A[J,JV]:=0;
                 M:=J-1;
                 for I:=M step 1 until J do A[I,J]:=0
           end
           READ DATA AND ACCUMULATE SUMS OF POWERS;
     10:   read P,Q,J;
           if J = 0 then go to 12 else go to 16;
     12:   NP:=NP+1;
           S:=(P-XMN)×ARX;
           YSQ:=YSQ+Q×Q;
```

```
      T:=1;
      K:=1;
      for J:=2 step 1 until MAXI do
      begin M:=J+1;
            for I:=M step 1 until J do
            begin if (K-MAXI) ≤ 0 then go to 13
                  else go to 14;
            13:   A[K,JV]:=A[K,JV]+T×Q;
                  K:=K+1;
            14:   T:=T×S;
                  A[I,J]:=A[I,J[+T
            end
      end;
      go to 10;
16:   print SCALED(4),SAMELINE,XMN,XRA,DIGIT(4),MIND,INC;
      D:=NP;
      A[1,1]:=D;
      N:=(MAXI-KMIN)/INC;
      if (NP-N) ≤ 0 then go to 17 else go to 18;
17:   N:=NP-1;
18:   MAXD:=N×INC+MIND;
      MAXI:=MAXD+1;
comment SPREAD TOTALS THROUGHOUT TOP HALF OF MATRIX;
      M1:=MAXI-2;
      for I:=KMIN step INC until M1 do
      begin for J:=I step INC until MAXI do
            begin if(J-I-2) ≥ 0 then go to 19 else go to 20;
            19:   L:=I+J;
                  K:=L/2;
                  L:=L-K;
                  A[I,J]:=A[K,L];
      20:   end
      end;
      print $MAX DEG?,SAMELINE,DIGITS(2),MAXD,$ NO POINTS ?,
            DIGITS(4),NP,$ RAW S OF S ?,SCALED(5),YSQ;
comment SET UP TO SOLVE EQUATIONS;
      for K:=KMIN step INC until MAXI do
      begin P:=1/A[K,K];
            A[K,K]:=A[K,JV]×P;
      comment SOLVE FOR RESIDUALS;
            YSQ:=YSQ-A[K,K]×A[K,JV];
            D:=D-1;
            if D ≤ 0 then go to 22 else go to 23;
      22:   Q:=0.9999₁₀99;
            go to 26;
      23:   if YSQ ≤ 0 then go to 24 else go to 25;
      24:   Q:=0;
            go to 26;
      25:   Q:=sqrt(YSQ/D);
      26:   J:=K-1;
            print $POLY DEG?,SAMELINE,DIGITS(2),J,
                  $ RESID S OF S ?,  SCALED(5),YSQ,
                  $ ERROR ?,Q;
comment SOLVE FOR COEFFICIENTS;
            M1:=K+INC;
            M:=MAXI-M1;
            if M ≥ 0 then go to 28 else go to 30;
      28:   for J:=M1 step INC until MAXI do
            begin A[J,K]:=A[K,J];
                  A[K,J]:=A[K,J]×P
            end;
      30:   L:=1;
            N:=KMIN;
            T:=1;
```

```
            for I:=1 step 1 until MAXI do
            begin if (N-I) = 0 then go to 31 else
                  if (N-I) < 0 then go to 4 else go to 40;
            31:   N:=N+INC;
                  if (I-K) = 0 then go to 37 else
                  if (I-K) < 0 then go to 33 else go to 32;
            32:   M1:=I;
            33:   P:=A[I,K];
                  if M ≥ 0 then go to 34 else go to 36;
            34:   for J:=M1 step INC until MAXI do
                  A[I,J]:=A[I,J]-A[K,J]×P;
            36:   A[I,K]:=A[I,JV]-A[K,K]×P;
                  go to JUMP[L];
            37:   L:=2;
            38:   J:=I-1;
                  R:=A[I,K]×T;
                  print SAMELINE,DIGITS(3),J,8 DEG COEF ?,
                        SCALED(5),R;
           40:    T:=T×ARX;
      41:   end;
      42:   JV:=K
            end
69:   end
```

input data

```
0
2
0
1
3
-.9
.5
0
-.7
.977
0
-.5
1.199
0
-.3
1.345
0
-.1
1.454
0
.1
1.541
0
.3
1.614
0
.5
1.676
0
.7
1.73
0
.9
1.779
0
0
0
1
```

output values

```
 0.000@+00 2.000@+00      0      1
MAX DEG 3 NO POINTS    10 RAW S OF S 2.0512@+01
POLY DEG  0 RESID S OF S 1.4262@+00 ST ERROR 3.9808@-01
    0 DEG COEF   1.3815@+00
POLY DEG  1 RESID S OF S 2.0672@-01 ST ERROR 1.6075@-01
    0 DEG COEF   1.3815@+00
    1 DEG COEF   6.0791@-01
POLY DEG  2 RESID S OF S 4.3332@-02 ST ERROR 7.8678@-02
    0 DEG COEF   1.5266@+00
    1 DEG COEF   6.0791@-01
    2 DEG COEF  -4.3977@-01
POLY DEG  3 RESID S OF S 9.2592@-03 ST ERROR 3.9284@-02
    0 DEG COEF   1.5266@+00
    1 DEG COEF   3.6462@-01
    2 DEG COEF  -4.3977@-01
    3 DEG COEF   4.1516@-01
```

6.2.5 A More Complex Example Including Input Data and Computed Output Values

Remarks: This example is written in FORTRAN II and translated into ALGOL.

The program, including the input data given, was run on a IBM 1620_{II}. The output values of this computation are presented.

The input/output statements are translated according to the rules of the NE computer (except that read and print appearing in the example are written with small letters). In order to run the translated program, it was changed due to the hardware representation and restrictions of the NE (this modified program is not given here) and fed, together with the data given, into the NE 803B. The output values of this computation are also presented.

In order to compare the output values from the two programs, one must know that the numbers of the NE output are rounded and that the numbers of the IBM output are truncated; thus a difference of ± 1 in the last digit is due to this procedure.

program

```
C CALCULATION OF THE TSAI-RADIATION-CORRECTION(PHYS.REV.122,1898(1961))
C CORR FOR A GIVEN DE, AN ELECTRON-ENERGY E, A SCATTERING ANGLE
C THETA AND AN ATOMIC NUMBER Z.
    1 FORMAT(I10,3F10.3,F15.5)
    2 FORMAT(33H NOTE BORN-APPROXIMATION VIOLATED)
    6 FORMAT(3F8.1,2F8.3,I7)
    3 FORMAT(27H TSAI-RADIATION-CORRECTION ////)
    4 FORMAT(46H      E      THETA      DE      DELTA      CORR      Z //)
      PUNCH3
      PUNCH 4
      READ 1,NZ,E,THETA,DE,GRM
      Z=NZ
      IF(NZ-8)10,10,11
   11 PUNCH 2
   10 A=THETA*3.1416/180.
      ZAHL=-1./(137.04*3.1416)
      EMEM=0.511*0.511
      ETA=1.+E*(1.-COSF(A))/GRM
```

```
      E3=E/ETA
      E4=E+GRM-E3
      BETA4=SQRTF(E4*E4-GRM*GRM)/E4
      QQ=(-4.*E*E*(SINF(0.5*A))**2)/ETA
      TERM1=ZAHL*(28./9.-(13./6.)*LOGF(-QQ/EMEM)+(LOGF(-QQ/EMEM)
     1 1.+2.*Z*LOGF(ETA))*(2.*LOGF(E/DE)-3.*LOGF(ETA)))
      TERM2=ZAHL*(F((E3-E)/E3)-Z*Z*LOGF(E4/GRM))
      TERM3=ZAHL*(Z*Z*LOGF(GRM/(ETA*DE))*((1./BETA4)*LOGF
     1 ((1.+BETA4)/(1.-BETA4))-2.)+(Z*Z/BETA4)*(0.5*LOGF((1.+BETA4)
     2 /(1.-BETA4))*LOGF((E4+GRM)/(2.*GRM))-F(-SQRTF((E4-GRM)/(E4+GRM))
     3 *SQRTF((1.+BETA4)/(1.-BETA4)))))
      TERM4=ZAHL*Z*(F(-(GRM-E3)/E)-F(GRM*(GRM-E3)/
     1 (2.*E3*E4-GRM*E))+F(2.*E3*(GRM-E3)/(2.*E3*E4-GRM*E))+LOGF(ABSF(
     2 (2.*E3*E4-GRM*E)/(E*(GRM-2.*E3))))*LOGF(GRM/(2.*E3)))
      TERM5=ZAHL*Z*(F(-(E4-E3)/E3)-F(GRM*(E4-E3)/(2.*E*E4-GRM*E3))
     1 +F(2.*E*(E4-E3)/(2.*E*E4-GRM*E3))+LOGF(ABSF((2.*E*E4-GRM*E3)
     2 /(E3*(GRM-2.*E))))*LOGF(GRM/(2.*E)))
      TERM6=ZAHL*Z*(F(-(GRM-E)/E)-F((GRM-E)/E)+F(2.*(GRM-E)/GRM)
     1 +LOGF(ABSF(GRM/(2.*E-GRM)))*LOGF(GRM/(2.*E)))
      TERM7=ZAHL*Z*(F(-(GRM-E3)/E3)-F((GRM-E3)/E3)+F(2.*(GRM-E3)/
     1 GRM)+LOGF(ABSF(GRM/(2.*E3-GRM)))*LOGF(GRM/(2.*E3)))
      TSAI=TERM1-TERM2+TERM3+TERM4-TERM5-TERM6+TERM7
      CORR=EXPF(TSAI)
      PUNCH 6,E,THETA,DE,TSAI,CORR,Z
      STOP
      END

C THIS IS THE SPENCE-FUNCTION CALCULATED FROM MITCHELL(PHIL. MAG.
C 40,351,(1949))
      FUNCTION F(X)
      PIPI=9.8696/6.
      IF(X+1.)101,102,102
  101 Y=1./(1.-Y)
      F=-PIPI+SUM(Y)-0.5*(LOGF(ABSF(1.-X)))**2
      RETURN
  102 IF(X)103,103,104
  103 Y=X/(X-1.)
      F=-SUM(Y)-0.5*(LOGF(ABSF(X-1.)))**2
      RETURN
  104 IF(X-0.5)105,103,106
  105 Y=X
      F=SUM(Y)
      RETURN
  106 IF(X-1.)107,108,108
  107 Y=1.-X
      F=PIPI-SUM(Y)-LOGF(X)*LOGF(1.-X)
      RETURN
  108 IF(X-2.)109,110,110
  109 Y=(X-1.)/X
      F=PIPI+SUM(Y)-0.5*LOGF(ABSF(X))*LOGF(((X-1.)**2)/ABSF(X))
      RETURN
  110 Y=1./X
      F=PIPI*2.-SUM(Y)-0.5*(LOGF(X))**2
      RETURN
      END
C END OF SUBPROGRAM F(X)

      FUNCTION SUM(Y)
      DO 10 N=2,8
      AN=N*N*(N+1)
   10 T=(Y-1.)**N/AN
      SUM=(Y*(3.+Y/4.+T)+2.*(1.-Y)*LOGF(1.-Y))/(1.+Y)
      RETURN
C THIS IS THE SUM TERM Y**N/(N*N(N+1))
      END
```

input data

```
1    900.        145.        13.1        938.213
```

output values

```
TSAI-RADIATION-CORRECTION

E          THETA          DE          DELTA          CORR          Z

900.0        145.0          13.1        -.155          .856          1
```

translated program

```
begin comment CALCULATION OF THE TSAI-RADIATION-CORRECTION
              (PHYS.REV.122,1898(1961)) CORR FOR A GIVEN DE,
              AN ELECTRON-ENERGY E, A SCATTERING ANGLE THETA
              AND AN ATOMIC NUMBER Z.;
integer NZ,N;
real Z,E,THETA,DE,A,ETA,GRM,EMEM,E3,E4,BETA4,QQ,ZAHL,TERM1,
    TERM2,TERM3,TERM4,TERM5,TERM6,TERM7,TSAI,CORR;
real procedure F(X);
    value X;
    real X;
    begin real PIPI,Y;
          real procedure SUM(Y);
              value Y;
              real Y;
              begin integer N;
                    real T,AN;
                    for N:=2 step 1 until 8 do
                    begin AN:=N×N×(N+1);
                          T:=(Y-1)↑N/AN
                    end;
                    SUM:=(Y×(3+Y/4+T)+2×(1-Y)×ln(1-Y))/(1+Y)
              end
              THIS IS THE SUM TERM Y↑N/(N×N(N+1));
              begin PIPI:=9.8696/6;
                    if (X+1) ≥ 0 then go to 102 else go to 101;
              101:  Y:=1/(1-X);
                    F:=-PIPI+SUM(Y)-0.5×ln(abs(1-X))↑2;
                    go to 211;
              102:  if X ≤ 0 then go to 103 else go to 104;
              103:  Y:=X/(X-1);
                    F:=-SUM(Y)-0.5×(ln(abs(X-1)))↑2;
                    go to 211;
              104:  if (X-0.5) = 0 then go to 103 else
                    if (X-0.5) < 0 then go to 105 else go to 106;
              105:  Y:=X;
                    F:=SUM(Y);
                    go to 211;
              106:  if (X-1) ≥ 0 then go to 108 else go to 107;
              107:  Y:=1-X;
                    F:=PIPI-SUM(Y)-ln(X)×ln(1-X);
                    go to 211;
              108:  if (X-2) ≥ 0 then go to 110 else go to 109;
              109:  Y:=(X-1)/X;
                    F:=PIPI+SUM(Y)-0.5×ln(abs(X))×ln(((X-1)↑2)/
                        abs(X));
                    go to 211;
              110:  Y:=1/X;
                    F:=PIPI×2-SUM(Y)-0.5×(ln(X))↑2;
          211 end
    end OF SUBPROGRAM F(X);
```

```
    print $ TSAI-RADIATION-CORRECTION $L4??;
    print $H  E        THETA        DE        DELTA        CORR        Z $L2??;
    read NZ,E,THETA,DE,GRM;
    Z:=NZ;
    if (NZ-8) ≤ 0 then go to 10 else go to 11;
11: print $ NOTE BORN-APPROXIMATION VIOLATED?;
10: A:=THETA×3.1416/180;
    ZAHL:=-1/(137.04×3.1416);
    EMEM:=0.511×0.511
    ETA:=1+E×(1-cos(A))/GRM;
    E3:=E/ETA;
    E4:=E+GRM-E3;
    BETA4:=sqrt(E4×E4-GRM×GRM)/E4;
    QQ:=(-4×E×E×(sin(0.5×A))↑2)/ETA;
    TERM1:=ZAHL×(28/9-(13/6)×ln(-QQ/EMEM)+(ln(-QQ/EMEM)
          -1+2×Z×ln(ETA))×(2×ln(E/DE)-3×ln(ETA)));
    TERM2:=ZAHL×(F((E3-E)/E3)-Z×Z×ln(E4/GRM));
    TERM3:=ZAHL×(Z×Z×ln(GRM/(ETA×DE))×((1/BETA4)×ln((1+BETA4)
          /(1-BETA4))-2)+(Z×Z/BETA4)×(0.5×ln((1+BETA4)/
          (1-BETA4))×ln((E4+GRM)/(2×GRM))-F(-sqrt((E4-GRM)/
          (E4+GRM))×sqrt((1+BETA4)/(1-BETA4)))));
    TERM4:=ZAHL×Z×(F(-(GRM-E3)/E)-F(GRM×(GRM-E3)/(2×E3×E4-GRM×E))
          +F(2×E3×(GRM-E3)/(2×E3×E4-GRM×E))+ln(abs((2×E3×E4-
          GRM×E)/(E×(GRM-2×E3))))×ln(GRM/(2×E3)));
    TERM5:=ZAHL×Z×(F(-(E4-E3)/E3)-F(GRM×(E4-E3)/(2×E×E4-GRM
          ×E3))+F(2×E×(E4-E3)/(2×E×E4-GRM×E3))+ln(abs((2×E×
          E4-GRM×E3)/(E3×(GRM-2×E))))×ln(GRM/(2×E)));
    TERM6:=ZAHL×Z×(F(-(GRM-E)/E)-F((GRM-E)/E)+F(2×(GRM-E)/
          GRM)+ln(abs(GRM/(2×E-GRM)))×ln(GRM/(2×E)));
    TERM7:=ZAHL×Z×(F(-(GRM-E3)/E3)-F((GRM-E3)/E3+F(2×(GRM-E3)/
          GRM)+ln(abs(GRM/(2×E3-GRM)))×ln(GRM/(2×E3)));
    TSAI:=TERM1-TERM2+TERM3+TERM4-TERM5-TERM6+TERM7;
    CORR:=exp(TSAI);
    print ALIGNED(5,1),SAMELINE,E,THETA,DE,ALIGNED(3,3),TSAI,
        CORR,  DIGITS(6),Z
end
```

input data

```
1
900
145
13.1
938.213
```

output values

```
TSAI-RADIATION-CORRECTION


E        THETA        DE        DELTA        CORR        Z

900.0      145.0        13.1    -0.154       0.857        1
```

Appendix I

Definition of ALGOL 60

(Published in *Numerische Mathematik* **4**, 420 (1963)
and *Communications of the Association for Computing Machinery*
6, No. 1 (1963))

REVISED REPORT ON THE ALGORITHMIC LANGUAGE ALGOL 60 *

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

Dedicated to the memory of William Turanski

* International Federation for Information Processing 1962.

SUMMARY

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

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

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

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

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

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

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

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

INTRODUCTION

Background

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

As a result of an informal meeting held at Mainz in November 1958, about forty interested persons from several European countries held an ALGOL implementation conference in Copenhagen in February 1959. A "hardware group" was formed for working cooperatively right down to the level of the paper

* "Preliminary Report—International Algebraic Language," *Communs Ass. comput. Mach.* **1**, No. 12 (1958), 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* **1**, 41–60, 1959.

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

January 1960 Conference

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

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

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

April 1962 Conference [Edited by M. WOODGER]

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

Authors	*Advisers*	*Observer*
F. L. BAUER	M. PAUL	W. L. VAN DER POEL
J. GREEN	R. FRANCIOTTI	(Chairman, IFIP TC 2.1
C. KATZ	P. Z. INGERMAN	Working Group
R. KOGON (representing J. W. BACKUS)		ALGOL)
P. NAUR		
K. SAMELSON	G. SEEGMÜLLER	
J. H. WEGSTEIN	R. E. UTMAN	
A. VAN WIJNGAARDEN		
M. WOODGER	P. LANDIN	

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

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

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

1. Side effects of functions.
2. The call by name concept.
3. **own**: static or dynamic.
4. For statement: static or dynamic.
5. Conflict between specification and declaration.

The authors of the ALGOL 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.

* *Editor's note:* The present edition follows the text which was approved by the Council of IFIP. Although it is not clear from the Introduction, the present version is the original report of the January 1960 Conference modified according to the agreements reached during the April 1962 Conference. Thus, the report mentioned here is incorporated in the present version. The modifications touch the original report in the following sections: Changes of text: 1 with footnote; 2.1 footnote; 2.3; 2.7; 3.3.3; 3.3.4.2; 4.1.3; 4.2.3; 4.2.4; 4.3.4; 4.7.3; 4.7.3.1; 4.7.3.3; 4.7.5.1; 4.7.5.4; 4.7.6; 5; 5.3.3; 5.3.5; 5.4.3; 5.4.4; 5.4.5. Changes of syntax: 3.4.1; 4.1.1; 4.2.1; 4.5.1.

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.

Reference language	*Publication language*
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 bracket $\langle\rangle$ represent metalinguistic variables whose values are sequences of symbols. The marks ::= and | (the latter with the meaning of **or**) are metalinguistic connectives. Any mark in a formula, which is not a variable or a connective, denotes itself (or the class of marks which are similar to it). Juxtaposition of marks and/or variables in a formula signifies juxtaposition of the sequences denoted. Thus, the formula above gives a recursive rule for the formation of values of the variable $\langle ab\rangle$. 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

* 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 Zürich ACM–GAMM conference." ICIP Paris, June 1959.

it with some value of the variable ⟨d⟩. If the values of ⟨d⟩ are the decimal digits, some values of ⟨ab⟩ are:

[(((*1(37(*
(12345(
(((
[*86*

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

Definition:

⟨empty⟩::=
(i.e. the null string of symbols).

2. Basic Symbols, Identifiers, Numbers, and Strings. Basic Concepts

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

⟨basic symbol⟩::=⟨letter⟩ | ⟨digit⟩ | ⟨logical value⟩ | ⟨delimiter⟩

2.1 Letters

⟨letter⟩::=*a*|*b*|*c*|*d*|*e*|*f*|*g*|*h*|*i*|*j*|*k*|*l*|*m*|*n*|*o*|*p*|*q*|*r*|*s*|*t*|*u*|*v*|*w*|*x*|*y*|*z*|
A|*B*|*C*|*D*|*E*|*F*|*G*|*H*|*I*|*J*|*K*|*L*|*M*|*N*|*O*|*P*|*Q*|*R*|*S*|*T*|*U*|*V*|*W*|*X*|*Y*|*Z*

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

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

2.2.1. *Digits.*

⟨digit⟩::=*0*|*1*|*2*|*3*|*4*|*5*|*6*|*7*|*8*|*9*

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

2.2.2. *Logical values.*

⟨logical value⟩::=**true** | **false**

The logical values have a fixed obvious meaning.

2.3. Delimiters

⟨delimiter⟩::=⟨operator⟩ | ⟨separator⟩ | ⟨bracket⟩ | ⟨declarator⟩ | ⟨specificator⟩
⟨operator⟩::=⟨arithmetic operator⟩ | ⟨relational operator⟩ | ⟨logical operator⟩ |
⟨sequential operator⟩

* It should be particularly noted that throughout the reference language underlining [in typewritten copy; bold face type in printed copy—*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. Within the present report [not included headings—*Ed.*] underlining [bold face—*Ed.*] will be used for no other purposes.

⟨arithmetic operator⟩::= + | − | × | / | ÷ | ↑
⟨relational operator⟩::= < | ≤ | = | ≥ | > | ≠
⟨logical operator⟩::= ≡ | ⊃ | ∨ | ∧ | ¬
⟨sequential operator⟩::= **go to** | **if** | **then** | **else** | **for** | **do**†
⟨separator⟩::= , | . | 10 | : | ; | := | ␣ | **step** | **until** | **while** | **comment**
⟨bracket⟩::= (|) | [|] | ' | ' | **begin** | **end**
⟨declarator⟩::= **own** | **Boolean** | **integer** | **real** | **array** | **switch** | **procedure**
⟨specificator⟩::= **string** | **label** | **value**

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

Typographical features such as blank space or change to a new line have no significance in the reference language. They may, however, be used freely for facilitating reading.

For the purpose of including text among the symbols of a program the following "comment" conventions hold:

The sequence of basic symbols:	is equivalent to
; **comment** ⟨any sequence not containing;⟩;	;
begin comment ⟨any sequence not containing;⟩;	**begin**
end ⟨any sequence not containing **end** or; or **else**⟩	**end**

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

2.4. Identifiers

2.4.1. *Syntax.*

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

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.*

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

† **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.

⟨decimal fraction⟩::=.⟨unsigned integer⟩
⟨exponent part⟩::=$_{10}$⟨integer⟩
⟨decimal number⟩::=⟨unsigned integer⟩ | ⟨decimal fraction⟩ |
⟨unsigned integer⟩ ⟨decimal fraction⟩
⟨unsigned number⟩::=⟨decimal number⟩ | ⟨exponent part⟩ |
⟨decimal number⟩ ⟨exponent part⟩
⟨number⟩::=⟨unsigned number⟩ | +⟨unsigned number⟩ | −⟨unsigned number⟩

2.5.2. *Examples.*

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

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

2.5.4. *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.*

⟨proper string⟩::=⟨any sequence of basic symbols not containing 'or'⟩ |
⟨empty⟩
⟨open string⟩::=⟨proper string⟩ | '⟨open string⟩' | ⟨open string⟩ ⟨open string⟩
⟨string⟩::='⟨open string⟩'

2.6.2. *Examples.*

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

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

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

2.7. Quantities, Kinds, and Scopes

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

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

2.8. Values and Types

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

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

The various "types" (**integer, real, Boolean**) basically denote properties of values. The types associated with syntactic units refer to the values of these units.

3. Expressions

In the language the primary constituents of the programs describing algorithmic processes are arithmetic, Boolean, and designational expressions. Constituents of these expressions, except for certain delimiters, are logical values, numbers, variables, function designators, and elementary arithmetic, relational, logical, and sequential operators. Since the syntactic definition of both variables and function designators contains expressions, the definition of expressions, and their constituents, is necessarily recursive.

⟨expression⟩::=⟨arithmetic expressions⟩ | ⟨Boolean expression⟩ |
⟨designational expression⟩

3.1. Variables

3.1.1. *Syntax.*

⟨variable identifier⟩::=⟨identifier⟩
⟨simple variable⟩::=⟨variable identifier⟩
⟨subscript expression⟩::=⟨arithmetic expression⟩
⟨subscript list::=⟨subscript expression⟩ | ⟨subscript list⟩,⟨subscript expression⟩
⟨array identifier⟩::=⟨identifier⟩
⟨subscripted variable⟩::=⟨array identifier⟩[⟨subscript list⟩]
⟨variable⟩::=⟨simple variable⟩ | ⟨subscripted variable⟩

3.1.2. *Examples.*

epsilon
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.*

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

⟨letter string⟩::=⟨letter⟩ | ⟨letter string⟩ ⟨letter⟩
⟨parameter delimiter⟩::=, |) ⟨letter string⟩:(
⟨actual parameter list⟩::=⟨actual parameter⟩ |
⟨actual parameter list⟩ ⟨parameter delimiter⟩ ⟨actual parameter⟩
⟨actual parameter part⟩::=⟨empty⟩ | (⟨actual parameter list⟩)
⟨function designator⟩::=⟨procedure identifier⟩ ⟨actual parameter part⟩

3.2.2. *Examples.* *sin*(*a* − *b*)
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 expression⟩ |
⟨if clause⟩ ⟨simple arithmetic expression⟩ **else** ⟨arithmetic expression⟩

3.3.2. *Examples.* Primaries:

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

Factors:

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

Terms:

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

Simple arithmetic expression:

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

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
$a \times sin(omega \times t)$
$0.57_{10}12 \times a[N \times (N - 1)/2,0]$
$(A \times arctan(y) + Z) \uparrow (7 + Q)$
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 is then the value of the first arithmetic expression following this Boolean (the largest arithmetic expression found in this position is understood). The construction:

else ⟨simple arithmetic expression⟩

is equivalent to the construction:

else if true then ⟨simple arithmetic expression⟩

3.3.4. *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 ⟨term⟩/⟨factor⟩ and ⟨term⟩ $\div$ ⟨factor⟩ both denote division, to be understood as a multiplication of the term by the reciprocal of the factor with due regard to the rules of precedence (cf. Section 3.3.5). Thus, for example

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

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:

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

(cf. Sections 3.2.4 and 3.2.5).

3.3.4.3. The operation ⟨factor⟩ $\uparrow$ ⟨primary⟩ denotes exponentiation, where the factor is the base and the primary is the exponent. Thus, for example,

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

while

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

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 $i > 0$:	$a \times a \times \ldots \times a$ (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 $a \neq 0$:	$1/(a \times a \times \ldots \times a)$ (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 $r > 0$:	0.0, of type **real**.
		if $r \leqslant 0$:	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.*

⟨relational operator⟩::= $<$ | $\leqslant$ | $=$ | $\geqslant$ | $>$ | $\neq$
⟨relational⟩::=⟨simple arithmetic expression⟩ ⟨relational operator⟩
⟨simple arithmetic expression⟩
⟨Boolean primary⟩::=⟨logical value⟩ | ⟨variable⟩ | ⟨function designator⟩ |
⟨relation⟩ | ⟨Boolean expression⟩ |
⟨Boolean secondary⟩::=⟨Boolean primary⟩ | $\neg$ ⟨Boolean primary⟩
⟨Boolean factor⟩::=⟨Boolean secondary⟩ |
⟨Boolean factor⟩ $\wedge$ ⟨Boolean secondary⟩
⟨Boolean term⟩::=⟨Boolean factor⟩ | ⟨Boolean term⟩ V ⟨Boolean factor⟩
⟨implication⟩::=⟨Boolean term⟩ | ⟨implication⟩ $\supset$ ⟨Boolean term⟩
⟨simple Boolean::=⟨implication⟩ | ⟨simple Boolean⟩$\equiv$⟨implication⟩
⟨Boolean expression⟩::=⟨simple Boolean⟩ |
⟨if clause⟩ ⟨simple Boolean⟩ **else** ⟨Boolean expression⟩

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** $h \leqslant c$
if if if a **then** b **else** c **then** d **else** f **then** g **else** $h < k$

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

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

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

The meaning of the logical operators $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\supset$ (implies), and $\equiv$ (equivalent), is given by the following function table:

b1	**false**	**false**	**true**	**true**
b2	**false**	**true**	**false**	**true**
$\neg$ *b1*	**true**	**true**	**false**	**false**
b1 $\wedge$ *b2*	**false**	**false**	**false**	**true**
b1 $\vee$ *b2*	**false**	**true**	**true**	**true**
b1 $\supset$ *b2*	**true**	**true**	**false**	**true**
b1 $\equiv$ *b2*	**true**	**false**	**false**	**true**

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

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

first: arithmetic expressions according to Section 3.3.5.
second: $< \leqslant = \geqslant > \neq$
third: $\neg$
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.*

⟨label⟩::=⟨identifier⟩ | ⟨unsigned integer⟩
⟨switch identifier⟩::=⟨identifier⟩
⟨switch designator⟩::=⟨switch identifier⟩ [⟨subscript expression⟩]
⟨simple designational expression⟩::=⟨label⟩ | ⟨switch designator⟩ |
(⟨designational expression⟩)
⟨designational expression⟩::=⟨simple designational expression⟩ |
⟨if clause⟩ ⟨simple designational expression⟩ **else** ⟨designational expression⟩

3.5.2. *Examples.* *17*
p9
Choose[*n* − *1*]
Town[**if** *y* < *0* **then** *N* **else** *N* + *1*]
if *Ab* < *c* **then** *17* **else** *q*[**if** *w* ⩽ *0* **then** *2* **else** *n*]

3.5.3. *Semantics.* A designational expression is a rule for obtaining a label of a statement (cf. Section 4. Statements). Again the principle of the evaluation is entirely analogous to that of arithmetic expressions (Section 3.3.3). In the general case the Boolean expressions of the if clauses will select a simple designational expression. If this is a label the desired result is already found. A switch designator refers to the corresponding switch declaration (cf. Section 5.3. Switch Declarations) and by the actual numerical value of its subscript expression

selects one of the designational expressions listed in the switch declaration by counting these from left to right. Since the designational expression thus selected may again be a switch designator this evaluation is obviously a recursive process.

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

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

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 sucessor explicitly, and shortened by conditional statements, which may cause certain statements to be skipped.

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

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

4.1. Compound Statements and Blocks

4.1.1. *Syntax.*

⟨unlabelled basic statement⟩::=⟨assignment statement⟩ | ⟨go to statement⟩ |
⟨dummy statement⟩ | ⟨procedure statement⟩
⟨basic statement⟩::=⟨unlabelled basic statement⟩ | ⟨label⟩:⟨basic statement⟩
⟨unconditional statement⟩::=⟨basic statement⟩ |
⟨compound statement⟩ | ⟨block⟩
⟨statement⟩::=⟨unconditional statement⟩ | ⟨conditional statement⟩ |
⟨for statement⟩
⟨compound tail⟩::=⟨statement⟩ **end** | ⟨statement⟩;⟨compound tail⟩
⟨block head⟩::=**begin** ⟨declaration⟩ | ⟨block head⟩;⟨declaration⟩
⟨unlabelled compound⟩::=**begin** ⟨compound tail⟩
⟨unlabelled block⟩::=⟨block head⟩;⟨compound tail⟩
⟨compound statement⟩::=⟨unlabelled compound⟩ |
⟨label⟩:⟨compound statement⟩
⟨block⟩::=⟨unlabelled block⟩ | ⟨label⟩:⟨block⟩
⟨program⟩::=⟨block⟩ | ⟨compound statement⟩

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

Compound statement:

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

Block:

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

It should be kept in mind that each of the 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.*

⟨left part⟩::=⟨variable⟩:= | ⟨procedure identifier⟩:=
⟨left part list⟩::=⟨left part⟩ | ⟨left part list⟩ ⟨left part⟩
⟨assignment statement⟩::=⟨left part list⟩ ⟨arithmetic expression⟩ |
⟨left part list⟩ ⟨Boolean expression⟩

4.2.2. *Examples.*

```
s: = p[0]: = n: = n + 1 + .
n: = n + 1
A: = B/C − v − q × S
S[v,k + 2]: = 3 − arctan(s × zeta)
V: = Q > Y Λ Z
```

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

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

4.2.3.2. The expression of the statement is evaluated.

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

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

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

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

4.3. Go to Statements

4.3.1. *Syntax.*

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

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

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

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

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

4.4. Dummy Statements

4.4.1. *Syntax.*

⟨dummy statement⟩::=⟨empty⟩

4.4.2. *Examples.*

L:
begin . . . . ; *John*: **end**

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

4.5. Conditional Statements

4.5.1. *Syntax.*

⟨if clause⟩::=**if**⟨Boolean expression⟩ **then**
⟨unconditional statement⟩::=⟨basic statement⟩ | ⟨compound statement⟩ | ⟨block⟩
⟨if statement⟩::=⟨if clause⟩ ⟨unconditional statement⟩
⟨conditional statement⟩::=⟨if statement⟩ | ⟨if statement⟩ **else** ⟨statement⟩ | ⟨if clause⟩ ⟨for statement⟩ | ⟨label⟩:⟨conditional statement⟩

4.5.2. *Examples.* **if** $x > 0$ **then** $n := n + 1$

if $v > u$ **then** $V: q := n + m$ **else go to** R

if $s < 0 \vee P \leqslant Q$ **then** AA: **begin if** $q < v$ **then** $a := v/s$ **else** $y := 2 \times a$ **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, i.e. the statement following the complete conditional statement. Thus the effect of the delimiter **else** may be described by saying that it defines the successor of the statement it follows to be the statement following the complete conditional statement.

The construction

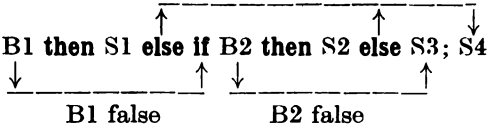
else ⟨unconditional statement⟩

is equivalent to

else if true then ⟨unconditional statement⟩

If none of the Boolean expressions of the if 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.**

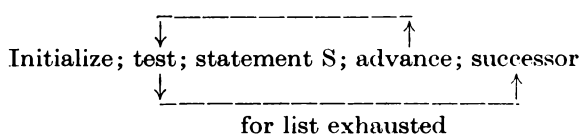
4.6. For Statements

4.6.1. *Syntax.*

⟨for list element⟩::=⟨arithmetic expression⟩ |
⟨arithmetic expression⟩ **step** ⟨arithmetic expression⟩ **until**
⟨arithmetic expression⟩ |
⟨arithmetic expression⟩ **while** ⟨Boolean expression⟩
⟨for list⟩::=⟨for list element⟩ | ⟨for list⟩, ⟨for list element⟩
⟨for clause⟩::=**for** ⟨variable⟩:=⟨for list⟩ **do**
⟨for statement⟩::=⟨for clause⟩ ⟨statement⟩ |
⟨label⟩:⟨for statement⟩

4.6.2. *Examples.* **for** q: = *1* **step** s **until** n **do** $A[q]$: = $B[q]$
for k: = *1,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.*

⟨actual parameter⟩::=⟨string⟩ | ⟨expression⟩ | ⟨array identifier⟩ |
⟨switch identifier⟩ | ⟨procedure identifier⟩
⟨letter string⟩::=⟨letter⟩ | ⟨string letter⟩ ⟨letter⟩
⟨parameter delimiter⟩::=, |) ⟨letter string⟩:(
⟨actual parameter list⟩::=⟨actual parameter⟩ |
⟨actual parameter list⟩ ⟨parameter delimiter⟩ ⟨actual parameter⟩
⟨actual parameter part⟩::=⟨empty⟩ | (⟨actual parameter list⟩)
⟨procedure statement⟩::=⟨procedure identifier⟩ ⟨actual parameter part⟩

4.7.2. *Examples.*

Spur (*A*) *Order:* (*7*) *Result to:* (*V*)
Transpose (*W,v* + *1*)
Absmax (*A,N,M,Yy,I,K*)
Innerproduct (*A*[*t,P,u*], *B*[*P*], *10*, *P*, *Y*)

These examples correspond to examples given in Section 5.4.2.

4.7.3. *Semantics.* A procedure statement serves to invoke (call for) the execution of a procedure body (cf. Section 5.4. Procedure Declarations). Where the procedure body is a statement written in ALGOL the effect of this execution will be

equivalent to the effect of performing the following operations on the program at the time of execution of the procedure statement:

4.7.3.1. *Value assignment (call by value).* All formal parameters quoted in the value part of the procedure declaration heading are assigned the values (cf. Section 2.8. Values and Types) of the corresponding actual parameters, these assignments being considered as being performed explicitly before entering the procedure body. The effect is as though an additional block embracing the procedure body were created in which these assignments were made to variables local to this fictitious block with types as given in the corresponding specifications (cf. Section 5.4.5). As a consequence, variables called by value are to be considered as non-local to the body of the procedure, but local to the fictitious block (cf. Section 5.4.3.)

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

4.7.5.1. If a string is supplied as an actual parameter in a procedure statement or function designator, whose defining procedure body is an ALGOL 60 statement (as opposed to non-ALGOL code, cf. Section 4.7.8), then this string can be used only 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 be derived only 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⟩;
⟨procdure 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 b 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 achieved (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 transformation. The summation is stopped as soon as tim
times in succession the absolute value of the terms of the transformed series are found
to be less than eps. Hence, one should provide a function fct with one integer argument,
an upper bound eps, and an integer tim. The output is the sum sum. euler is partic-
ularly 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])) Λ (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 = fk(x, y1, y2, . . . , yn)(k = 1,2, . . . n)
of differential equations with the method of Runge-Kutta with automatic search for
appropriate length of integration step. Parameters are : The initial values x and y[k]
for x and the unknown functions yk(x). The order n of the system. The procedure
FKT(x, y, n, z) which represents the system to be integrated, i.e. the set of functions fk.
The tolerance values eps and eta which govern the accuracy of the numerical integra-
tion. The end of the integration interval xE. The output parameter yE which re-
presents the solution at x = xE. The Boolean variable fi, which must always be
given the value true for an isolated or first entry into RK. If, however, the functions y
must be available at several meshpoints x0, x1, . . . , xn, then the procedure must be
called repeatedly (with x = xk, xE = xk+1, for k = 0, 1, . . . , n − 1) and then the
later calls may occur with fi = false which saves computing time. The input para-
meters of FKT must be x, y, n, the output parameter z represents the set of derivatives
z[k] = fk(x, y[1], y[2], . . . , y[n]) for x and the actual y's. A procedure comp enters
as a 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 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;
```

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

```
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 expo-
    nents of these quantities have been made equal to the largest of the exponents
    of the originally given parameters a, b, c;
    x: = x3; if out then go to DD;
    for k: = 1 step 1 until n do y[k]: = y3[k];
    if s = 5 then begin s: = 0; H: = 2 × H end if;
    s: = s + 1; go to AA;
CC: H: = 0.5 × H; out: = false; x1 = x2;
    for k: = 1 step 1 until n do y1[k]: = y2[k];
    go to BB;
DD: for k: = 1 step 1 until n do yE[k]: = y3[k]
end RK
```

ALPHABETIC INDEX OF DEFINITIONS OF CONCEPTS AND SYNTACTIC UNITS

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

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

The basic symbols represented by signs other than underlined (*bold faced. Publishers remark*) words have been collected at the beginning. The examples have been ignored in compiling the index.

+ see: plus
− see: minus
× see: multiply
/ ÷ see: divide
↑ see: exponentiation
< ≤ = ≥ > ≠ see: ⟨relational operator⟩
≡ ⊃ V Λ ⊅ 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
comma, , synt 2.3, 3.1.1, 3.2.1, 4.6.1, 4.7.1, 5.1.1, 5.2.1, 5.3.1, 5.4.1
comment, synt 2.3
comment convention, text 2.3
⟨compound statement⟩, def 4.1.1 synt 4.5.1 text 1
⟨compound tail⟩, def 4.1.1
⟨conditional statement⟩, def 4.5.1 synt 4.1.1 text 4.5.3
⟨decimal fraction⟩, def 2.5.1
⟨decimal number⟩, def 2.5.1 text 2.5.3
decimal point ., synt 2.3, 2.5.1
⟨declaration⟩, def 5 synt 4.1.1 text 1, 5 (complete section)
⟨declarator⟩, def 2.3
⟨delimiter⟩, def 2.3 synt 2
⟨designational expression⟩, def 3.5.1 synt 3, 4.3.1, 5.3.1 text 3.5.3

⟨digit⟩, def 2.2.1 synt 2, 2.4.1, 2.5.1
dimension, text 5.2.3.2
divide / ÷, synt 2.3, 3.3.1 text 3.3.4.2
do, synt 2.3, 4.6.1
⟨dummy statement⟩, def 4.4.1 synt 4.1.1 text 4.4.3

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

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

go to, synt 2.3, 4.3.1
⟨go to statement⟩, def 4.3.1 synt 4.1.1 text 4.3.3

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

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

minus −, synt 2.3, 2.5.1, 3.3.1 text 3.3.4.1
multiply ×, synt 2.3, 3.3.1 text 3.3.4.1
⟨multiplying operator⟩, def 3.3.1

non-local, text 4.1.3

⟨number⟩, def 2.5.1 text 2.5.3, 2.5.4

⟨open string⟩, def 2.6.1
⟨operator⟩, def 2.3
own, synt 2.3, 5.1.1 text 5, 5.2.5

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

quantity, text 2.7

real, synt 2.3, 5.1.1 text 5.1.3
⟨relation⟩, def 3.4.1 text 3.4.5
⟨relational operator⟩, def 2.3, 3.4.1

scope, text 2.7
semicolon;, synt 2.3, 4.1.1, 5.4.1
⟨separator⟩, def 2.3
⟨sequential operator⟩, def 2.3
⟨simple arithmetic expression⟩, def 3.3.1 text 3.3.3
⟨simple Boolean⟩, def 3.4.1
⟨simple designational expression⟩, def 3.5.1
⟨simple variable⟩, def 3.1.1 synt 5.5.1 text 2.4.3
space ␣, synt 2.3 text 2.3, 2.6.3
⟨specification part⟩, def 5.4.1 text 5.4.5
⟨specificator⟩, def 2.3
⟨specifier⟩, def 5.4.1
standard function, text 3.2.4, 3.2.5
⟨statement⟩, def 4.1.1, synt 4.5.1, 4.6.1, 5.4.1 text 4 (complete section)
statement bracket see: **begin end**
step, synt 2.3, 4.6.1 text 4.6.4.2
string, synt 2.3, 5.4.1
⟨string⟩, def 2.6.1 synt 3.2.1, 4.7.1 text 2.6.3
string quotes ‘ ’, synt 2.3, 2.6.1, text 2.6.3
subscript, text 3.1.4.1
subscript bound, text 5.2.3.1
subscript brackets[], synt 2.3, 3.1.1, 3.5.1, 5.2.1
⟨subscripted variable⟩, def 3.1.1 text 3.1.4.1
⟨subscript expression⟩, def 3.1.1 synt 3.5.1
⟨subscript list⟩, def 3.1.1
successor, text 4
switch, synt 2.3, 5.3.1, 5.4.1
⟨switch declaration⟩, def 5.3.1 synt 5 text 5.3.3
⟨switch designator⟩, def 3.5.1 text 3.5.3

⟨switch identifier⟩, def 3.5.1 synt 3.2.1, 4.7.1, 5.3.1
⟨switch list⟩, def 5.3.1

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

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

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

while, synt 2.3, 4.6.1 text 4.6.4.3

Appendix II

Definition of FORTRAN

(Published in *Communications of the Association for Computing Machinery* **7**, No. 10, 590 (1964))

HISTORY AND SUMMARY OF FORTRAN STANDARDIZATION DEVELOPMENT FOR THE ASA

By

W. P. Heising

THE American Standards Association (ASA) Sectional Committee X3 for Computers and Information Processing was established in 1960 under the sponsorship of the Business Equipment Manufacturers Association. ASA X3 in turn established an X3.4 Sectional Subcommittee to work in the area of common programming language standards. On May 17, 1962, X3.4 established by resolution a working group, X3.4.3-FORTRAN to develop American Standard FORTRAN proposals.

RESOLVED:

That X3.4 form a FORTRAN Working Group, to be known as X3.4.3-FORTRAN, with the

Scope. To develop proposed standards of FORTRAN language.

Organization. Shall contain a Policy Committee and a Technical Committee. The Policy Committee will be responsible to X3.4 for the Working Group's mission being accomplished. It will determine general policy, such as language content, and direct the Technical Committee.

Policy Committee Membership. Will be determined by the X3.4 Steering Committee subject to written guidelines which may be amended later and including the following:

(*a*) For each FORTRAN implementation in active development or use, one sponsor voting representative and one user voting representative are authorized,

(*b*) A representative who is inactive may be dropped.

(*c*) Associate members, not entitled to vote but entitled to participate in discussion, are authorized.

Technical Committee. Will develop proposed standards of FORTRAN language under the Policy Committee direction. The Technical Committee will conduct investigations and make reports to the Policy Committee.

On June 25, 1962, invitations to an organizational meeting of X3.4.3 were sent to manufacturers and user groups who might be interested in participating in the development of FORTRAN standards. The first meeting was held August 13–14, 1962, in New York City. X3.4.3 decided to proceed because (1) FORTRAN standardization was needed, and (2) a sufficiently wide representation of interested persons was participating.

A resolution on objectives was adopted unanimously on August 14, 1962.

The objective of the X3.4.3 Working Group of ASA is to produce a document or documents which will define the ASA Standard or Standards for the FORTRAN language. The resulting standard language will be clearly and recognizably related to that language, with its variations, which has been called FORTRAN in the past. The criteria used to consider and evaluate various language elements will include (not in order of importance):

(*a*) ease of use by humans;
(*b*) compatibility with past FORTRAN use;
(*c*) scope of application;
(*d*) potential for extension;
(*e*) facility of implementation, i.e. compilation and execution efficiency.

The FORTRAN standard will facilitate machine-to-machine transfer of programs written in ASA Standard FORTRAN. The Standard will serve as a

reference document both for users who wish to achieve this objective and for manufacturers whose programming products will make it possible. The content and method of presentation of the standard will recognize this purpose."

It was the consensus of the group that (1) there was definite interest in developing a standard corresponding to what is popularly known as FORTRAN IV, and (2) there was interest in developing for small and intermediate computers a FORTRAN standard near the power of FORTRAN II, however suitably modified to be compatible with the associated FORTRAN IV. Accordingly, two Technical Committees, designated X3.4.3-IV and X3.4.3-II respectively, were established to create drafts. Most of the detailed work in developing drafts has been done by technical committees.

The X.3.4.3-II Technical Committee completed and approved a draft in May 1963. A Technical Fact Finding Committee was appointed and reported in August 1964 on a comparison of the X3.4.3-II approved draft and an approved working draft of the X3.4.3-IV Technical Committee. This brought to light stylistic, terminological, and content differences and conflicts. In April 1964 the X3.4.3-IV Technical Committee completed a draft of FORTRAN. In June 1964 X3.4.3 received and compared the two drafts and (1) resolved conflicts in content, and (2) resolved the conflicting style and terminology. This was accomplished by recasting the X3.4.3-II document to reflect the style of the X3.4.3-IV document while retaining the original content. To reduce confusion, X.3.4.3 decided to call the languages Basic FORTRAN and FORTRAN.

The following working documents have been produced by a Subcommittee of the American Standards Association Sectional Committee X3, Computers and Information Processing, in its efforts to develop a proposed American Standard. In order that the final version of the proposed American Standard reflect the largest public consensus, X3 has authorized publication of these documents to elicit comment, criticism, and general public reaction, with the understanding that such working documents are intermediate results in the standardization process and are subject to change, modification, or withdrawal in part or in whole. Correspondence about the documents should be addressed to the X3 Secretary, BEMA, 235 East 42nd Street, New York, N.Y. 10017.—R.V.S.

FORTRAN*

CONTENTS

1. Introduction

2. Basic Terminology

3. Program Form
3.1. The FORTRAN character set
3.2. Lines
3.3. Statements
3.4. Statement label
3.5. Symbolic names
3.6. Ordering of characters

4. Data Types
4.1. Data types association
4.2. Data type properties

5. Data and Procedure Identification
5.1. Data and procedure names
5.1.1. Constants
5.1.2. Variables
5.1.3. Array
5.1.4. Procedures
5.2. Function reference
5.3. Type rules for data and procedure identifiers
5.4. Dummy arguments

6. Expressions
6.1. Arithmetic expressions
6.2. Relational expressions
6.3. Logical expressions
6.4. Evaluation of expressions

7. Statements
7.1. Executable statements
7.1.1. Assignment statements
7.1.2. Control statements
7.1.2.1. GO TO statements
7.1.2.2. Arithmetic IF statement
7.1.2.3. Logical IF statement
7.1.2.4. CALL statement
7.1.2.5. RETURN statement
7.1.2.6. CONTINUE statement
7.1.2.7. Program control statements
7.1.2.8. DO statement
7.1.3. Input/Output statements
7.1.3.1. READ and WRITE statements
7.1.3.2. Auxiliary Input/Output statements
7.1.3.3. Printing of formatted records
7.2. Nonexecutable statements
7.2.1. Specification statements
7.2.1.1. Array declarator
7.2.1.2. DIMENSION statement
7.2.1.3. COMMON statement
7.2.1.4. EQUIVALENCE statement
7.2.1.5. EXTERNAL statement
7.2.1.6. Type statement
7.2.2. Data initialization statement
7.2.3. FORMAT statement

8. Procedures and Subprograms
8.1. Statement functions
8.2. Intrinsic functions and their reference
8.3. External functions
8.4. Subroutine
8.5. Block data subprogram

9. Programs
9.1. Program components
9.2. Normal execution sequence

10. Intra- and Interprogram Relationships
10.1. Symbolic names
10.2. Definition
10.3. Definition requirements for use of entities

* Popularly known as FORTRAN IV. (H.B.)

1. INTRODUCTION

1.1. PURPOSE. This specification establishes the form for and the interpretation of programs expressed in the FORTRAN language for the purpose of promoting a high degree of interchangeability of such programs for use on a variety of automatic data processing systems. A processor shall conform to this specification provided it accepts, and interprets as specified, at least those forms and relationships described herein.

Insofar as the interpretation of the form and relationships described are not affected, any statement of requirement could be replaced by a statement expressing that the specification does not provide an interpretation unless the requirement is met. Further, any statement of prohibition could be replaced by a statement expressing that the specification does not provide an interpretation when the prohibition is violated.

1.2. SCOPE. This specification establishes:

1. The form of a program written in the FORTRAN language.
2. The form of writing input data to be processed by such a program operating on automatic data processing systems.
3. Rules for interpreting the meaning of such a program.
4. The form of the output data resulting from the use of such a program on automatic data processing systems, provided that the rules of interpretation establish an interpretation.

This specification does not prescribe:

1. The mechanism by which programs are transformed for use on a data processing system (the combination of this mechanism and data processing system is called a processor).
2. The method of transcription of such programs or their input or output data to or from a data processing medium.
3. The manual operations required for set-up and control of the use of such programs on data processing equipment.
4. The results when the rules for interpretation fail to establish an interpretation of such a program.
5. The size or complexity of a program that will exceed the capacity of any specific data processing system or the capability of a particular processor.
6. The range or precision of numerical quantities.

2. BASIC TERMINOLOGY

This section introduces some basic terminology and some concepts. A rigorous treatment of these is given in later sections. Certain assumptions concerning the meaning of grammatical signs and particular words are presented.

A program that can be used as a self-contained computing procedure is called an *executable program* (9.1.6).

An executable program consists of precisely one main program and possibly one or more subprograms (9.1.6).

A *main program* is a set of statements and comments not containing a FUNCTION, SUBROUTINE, or BLOCK DATA statement (9.1.5).

A *subprogram* is similar to a main program but is headed by a BLOCK DATA, FUNCTION, or SUBROUTINE statement. A subprogram headed by a BLOCK DATA statement is called a specification subprogram. A subprogram headed by

a FUNCTION or SUBROUTINE statement is called a procedure subprogram (9.1.3, 9.1.4).

The term *program unit* will refer to either a main program or subprogram (9.1.7).

Any program unit except a specification subprogram may reference an *external procedure* (Section 9).

An external procedure that is defined by FORTRAN statements is called a *procedure subprogram*. External procedures also may be defined by other means. An external procedure may be an external function or an external subroutine. An external function defined by FORTRAN statements headed by a FUNCTION statement is called a *function subprogram*. An external subroutine defined by FORTRAN statements headed by a SUBROUTINE statement is called a *subroutine subprogram* (Sections 8 and 9).

Any program unit consists of *statements* and *comments*. A statement is divided into physical sections called *lines*, the first of which is called an *initial line*, and the rest of which are called *continuation lines* (3.2).

There is a type of line called a comment that is not a statement and merely provides information for documentary purposes (3.2).

The statements in FORTRAN fall into two broad classes—executable and nonexecutable. The executable statements specify the action of the program while the nonexecutable statements describe the use of the program, the characteristics of the operands, editing information, statement functions, or data arrangement (7.1, 7.2).

The syntactic elements of a statement are *names* and *operators*. Names are used to reference objects such as data or procedures. Operators, including the imperative verbs, specify action upon named objects.

One class of name, the *array name*, deserves special mention. The name and the dimensions of the array of values denoted by the array name are declared prior to use. An array name may be used to identify an entire array. An array name qualified by a subscript may be used to identify a particular element of the array (5.1.3).

Data names and the arithmetic (or logical) operators may be connected into arithmetic (or logical) expressions that develop values. These values are derived by performing the specified operations on the named data (Section 6).

The identifiers used in FORTRAN are names and numbers. Data are named. Procedures are named. Statements are labelled with numbers. Input/output units are numbered or identified by a name whose value is the numerical unit designation (Sections 3, 6, 7).

At various places in this document there are statements with associated lists of entries. In all such cases the list is assumed to contain at least one entry unless an explicit exception is stated. As an example, in the statement

$$\text{SUBROUTINE } s(a_1, a_2, \ldots a_n)$$

it is assumed that at least one symbolic name is included in the list within parentheses. A *list* is a set of identifiable elements, each of which is separated from its successor by a comma. Further, in a sentence a plural form of a noun will be assumed to also specify the singular form of that noun as a special case when the context of the sentence does not prohibit this interpretation.

The term *reference* is used as a verb with special meaning as defined in Section 5.

3. PROGRAM FORM

Every program unit is constructed of characters grouped into lines and statements.

3.1. THE FORTRAN CHARACTER SET. A program unit is written using the following characters: 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, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, and:

Character	*Name of Character*
	Blank
=	Equals
+	Plus
−	Minus
*	Asterisk
/	Slash
(	Left Parenthesis
)	Right Parenthesis
,	Comma
.	Decimal Point
$	Currency Symbol

The order in which the characters are listed does not imply a collating sequence.

3.1.1. *Digits.* A digit is one of the ten characters: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Unless specified otherwise, a string of digits will be interpreted in the decimal base number system when a number system base interpretation is appropriate.

An octal digit is one of the eight characters: 0, 1, 2, 3, 4, 5, 6, 7. These are only used in the STOP (7.1.2.7.1) and PAUSE (7.1.2.7.2) statements.

3.1.2. *Letters.* A letter is one of the twenty-six characters: 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.

3.1.3. *Alphanumeric Characters.* An alphanumeric character is a letter or a digit.

3.1.4. *Special Characters.* A special character is one of the eleven characters blank, equals, plus, minus, asterisk, slash, left parenthesis, right parenthesis, comma, decimal point, and currency symbol.

3.1.4.1. *Blank Character.* With the exception of the uses specified (3.2.2, 3.2.3, 3.2.4, 4.2.6, 5.1.1.6, 7.2.3.6, and 7.2.3.8), a blank character has no meaning and may be used freely to improve the appearance of the program subject to the restriction on continuation lines in 3.3.

3.2. LINES. A line is a string of 72 characters. All characters must be from the FORTRAN character set except as described in 5.1.1.6 and 7.2.3.8.

The character positions in a line are called columns and are consecutively numbered 1, 2, 3, . . . , 72. The number indicates the sequential position of a character in the line starting at the left and proceeding to the right.

3.2.1. *Comment Line.* The letter C in column 1 of a line designates that line as a comment line. A comment line must be immediately followed by an initial line, another comment line, or an end line.

A comment line does not affect the program in any way and is available as a convenience for the programmer.

3.2.2. *End Line.* An end line is a line with the character blank in columns 1 through 6, the characters E, N, and D, once each and in that order, in columns 7 through 72, preceded by, interspersed with, or followed by the character blank.

The end line indicates to the processor the end of the written description of a program unit (9.1.7). Every program unit must physically terminate with an end line.

3.2.3. *Initial Line.* An initial line is a line that is neither a comment line nor an end line and that contains the digit 0 or the character blank in column 6. Columns 1 through 5 contain the statement label or each contains the character blank.

3.2.4. *Continuation Line.* A continuation line is a line that contains any character other than the digit 0 or the character blank in column 6, and does not contain the character C in column 1.

A continuation line may only follow an initial line or another continuation line.

3.3. STATEMENTS. A statement consists of an initial line optionally followed by up to nineteen ordered continuation lines. The statement is written in columns 7 through 72 of the lines. The order of the characters in the statement is columns 7 through 72 of the initial line followed, as applicable, by columns 7 through 72 of the first continuation line, columns 7 through 72 of the next continuation line, etc.

3.4. STATEMENT LABEL. Optionally, a statement may be labelled so that it may be referred to in other statements. A statement label consists of from one to five digits. The value of the integer represented is not significant but must be greater than zero. The statement label may be placed anywhere in columns 1 through 5 of the initial line of the statement. The same statement label may not be given to more than one statement in a program unit. Leading zeros are not significant in differentiating statement labels.

3.5. SYMBOLIC NAMES. A symbolic name consists of from one to six alphanumeric characters, the first of which must be alphabetic. See 10.1 through 10.1.10 for a discussion of classification of symbolic names and restrictions on their use.

3.6. ORDERING OF CHARACTERS. An ordering of characters is assumed within a program unit. Thus, any meaningful collection of characters that constitutes names, lines, and statements exists as a totally ordered set. This ordering is imposed by the character position rule of 3.2 (which orders characters within lines) and the order in which lines are presented for processing.

4. DATA TYPES

Six different types of data are defined. These are integer, real, double precision, complex, logical, and Hollerith. Each type has a different mathematical significance and may have different internal representation. Thus, the data type has a significance in the interpretation of the associated operations with which a datum is involved. The data type of a function defines the type of the datum it supplies to the expression in which it appears.

4.1. DATA TYPE ASSOCIATION. The name employed to identify a datum or function carries the data type association. The form of the string representing a constant defines both the value and the data type.

A symbolic name representing a function, variable, or array must have only a single data type association for each program unit. Once associated with a particular data type, a specific name implies that type for any differing usage of that symbolic name that requires a data type association throughout the program unit in which it is defined.

Data type may be established for a symbolic name by declaration in a type-statement (7.2.1.6) for the integer, real, double precision, complex, and logical

types. This specific declaration overrides the implied association available for integer and real (5.3).

There exists no mechanism to associate a symbolic name with the Hollerith data type. Thus data of this type, other than constants, are identified under the guise of a name of one of the other types.

4.2. DATA TYPE PROPERTIES. The mathematical and the representation properties for each of the data types are defined in the following sections. For real, double precision, and integer data, the value zero is considered neither positive nor negative.

4.2.1. *Integer Type.* An integer datum is always an exact representation of an integer value. It may assume positive, negative, and zero values. It may only assume integral values.

4.2.2. *Real Type.* A real datum is a processor approximation to the value of a real number. It may assume positive, negative, and zero values.

4.2.3. *Double Precision Type.* A double precision datum is a processor approximation to the value of a real number. It may assume positive, negative, and zero values. The degree of approximation, though undefined, must be greater than that of type real.

4.2.4. *Complex Type.* A complex datum is a processor approximation to the value of a complex number. The representation of the approximation is in the form of an ordered pair of real data. The first of the pair represents the real part and the second, the imaginary part. Each part has, accordingly, the same degree of approximation as for a real datum.

4.2.5. *Logical Type.* A logical datum may assume only the truth values of true or false.

4.2.6. *Hollerith Type.* A Hollerith datum carries symbolic information (as opposed to a numeric or logical value). The symbolic information may consist of any symbol combination capable of representation in the processor. The representation for blank is a valid and significant character in a Hollerith datum.

5. DATA AND PROCEDURE IDENTIFICATION

Names are employed to reference or otherwise identify data and procedures.

The term *reference* is used to indicate an identification of a datum implying that the current value of the datum will be made available during the execution of the statement containing the reference. If the datum is identified but not necessarily made available the datum is said to be *named.* One case of special interest in which the datum is named is that of assigning a value to a datum, thus defining or redefining the datum.

The term, reference, is used to indicate an identification of a procedure implying that the actions specified by the procedure will be made available.

A complete and rigorous discussion of reference and definition, including redefinition, is contained in Section 10.

5.1. DATA AND PROCEDURE NAMES. A data name identifies a constant, a variable, an array or array element, or a block (7.2.1.3). A procedure name identifies a function or a subroutine.

5.1.1. *Constants.* A constant is a name that references a value or symbolic information derived from the name. A constant may not be redefined.

An integer, real, or double precision constant is said to be signed when it is immediately preceded by a plus or minus. Also, for these types, an optionally signed constant is either a constant or a signed constant.

5.1.1.1. *Integer Constant.* An integer constant is formed by a nonempty string of digits. The datum formed this way is interpreted as the value represented by the digit string.

5.1.1.2. *Real Constant.* A basic real constant is formed by an integer part, a decimal point, and a decimal fraction part in that order. Both the integer part and the decimal fraction part are formed by a string of digits; either one of these strings may be empty, but not both. The datum formed this way is interpreted as representing a value that is an approximation to the number represented by the integer and fraction parts.

A decimal exponent is formed by the letter E followed by an optionally signed integer constant. This exponent is interpreted as a multiplier (to be applied to the constant immediately preceding it) that is an approximation to ten raised to the power specified by the field following the E.

A real constant is a basic real constant, a basic real constant followed by a decimal exponent, or an integer constant followed by a decimal exponent.

5.1.1.3. *Double Precision Constant.* A double precision exponent is formed and interpreted identically to a decimal exponent except that the letter D is used instead of the letter E.

A double precision constant is a basic real constant followed by a double precision exponent or an integer constant followed by a double precision exponent.

5.1.1.4. *Complex Constant.* A complex constant is formed by an ordered pair of optionally signed real constants, separated by a comma, and enclosed within parentheses. The datum formed this way is interpreted as an approximation to the complex number represented by the pair.

5.1.1.5. *Logical Constant.* A logical constant is formed as one of the strings TRUE. or .FALSE. ; these are interpreted as representing the truth values of true and false, respectively.

5.1.1.6. *Hollerith Constant.* A Hollerith constant is formed by an integer constant (whose value n is greater than zero) followed by the letter H , followed by exactly n characters. Any n characters capable of representation by the processor may follow the H . However, the differing character sets of different processors may cause the interpretation of these constants to vary. The character blank is significant in a Hollerith constant.

This constant form is only defined for use in the argument list of a CALL statement and in the data initialization statement.

5.1.2. *Variable.* A variable is a datum that is identified by a symbolic name (3.5). Such a datum may be referenced and defined.

5.1.3. *Array.* An array is an ordered set of data of one, two, or three dimensions. An array is identified by a symbolic name. Identification of the entire ordered set is achieved via use of the array name.

5.1.3.1. *Array Element.* An array element is one of the members of the set of data of an array. An array element is identified by immediately following the array name with a qualifier, called a subscript, which points to the particular element of the array.

An array element may be referenced and defined.

5.1.3.2. *Subscript.* A subscript is formed by a parenthesized list of subscript expressions. Each subscript expression is separated by a comma from its successor, if there is a successor. The number of subscript expressions must correspond to the declared dimensionality (7.2.1.1), except in an EQUIVALENCE statement (7.2.1.4). Following evaluation of all of the subscript expressions, the array element successor function (7.2.1.1.1) determines the identified array element.

5.1.3.3. *Subscript Expressions.* A subscript expression is formed from one of the following constructs:

$$c*v+k$$
$$c*v-k$$
$$c*v$$
$$v+k$$
$$v-k$$
$$v$$
$$k$$

where c and k are integer constants and v is an integer variable reference. See Section 6 for a discussion of evaluation of expressions and 10.2.8 and 10.3 for requirements that apply to the use of a variable in a subscript.

5.1.4. *Procedures.* A procedure (Section 8) is identified by a symbolic name. A procedure is a statement function, an intrinsic function, a basic external function, an external function, or an external subroutine. Statement functions, intrinsic functions, basic external functions, and external functions are referred to as functions or function procedures; external subroutines as subroutines or subroutine procedures.

A function supplies a result to be used at the point of reference; a subroutine does not. Functions are referenced in a manner different from subroutines.

5.2. FUNCTION REFERENCE. A function reference consists of the function name followed by an actual argument list enclosed in parentheses. If the list contains more than one argument, the arguments are separated by commas. The allowable forms of function arguments are given in Section 8.

See 10.2.1 for a discussion of requirements that apply to function references.

5.3. TYPE RULES FOR DATA AND PROCEDURE IDENTIFIERS. The type of a constant is implicit in its name.

There is no type associated with a symbolic name that identifies a subroutine or a block.

A symbolic name that identifies a variable, an array, or a statement function may have its type specified in a type-statement. In the absence of an explicit declaration, the type is implied by the first character of the name: I, J, K, L, M, and N imply type integer; any other letter implies type real.

A symbolic name that identifies an intrinsic function or a basic external function when it is used to identify this designated procedure, has a type associated with it as specified in Tables 3 and 4.

In the program unit in which an external function is referenced its type definition is defined in the same manner as for a variable and an array. For a function subprogram, type is specified either implicitly by its name or explicitly in the FUNCTION statement.

The same type is associated with an array element as is associated with the array name.

5.4. DUMMY ARGUMENTS. A dummy argument of an external procedure identifies a variable, array, subroutine, or external function.

When the use of an external function name is specified, the use of a dummy argument is permissible if an external function name will be associated with that dummy argument. (Section 8.)

When the use of an external subroutine name is specified, the use of a dummy argument is permissible if an external subroutine name will be associated with that dummy argument.

When the use of a variable or array element reference is specified, the use of a

dummy argument is permissible if a value of the same type will be made available through argument association.

Unless specified otherwise, when the use of a variable, array, or array element name is specified, the use of a dummy argument is permissible provided that a proper association with an actual argument is made.

The process of argument association is discussed in Sections 8 and 10.

6. EXPRESSIONS

This section gives the formation and evaluation rules for arithmetic, relational, and logical expressions. A relational expression appears only within the context of logical expressions. An expression is formed from elements and operators. See 10.3 for a discussion of requirements that apply to the use of certain entities in expressions.

6.1. ARITHMETIC EXPRESSIONS. An arithmetic expression is formed with arithmetic operators and arithmetic elements. Both the expression and its constituent elements identify values of one of the types integer, real, double precision, or complex. The arithmetic operators are:

Operator	*Representing*
+	Addition, positive value (zero + element)
−	Subtraction, negative value (zero − element)
*	Multiplication
/	Division
**	Exponentiation

The arithmetic elements are primary, factor, term, signed term, simple arithmetic expression, and arithmetic expression.

A primary is an arithmetic expression enclosed in parentheses, a constant, a variable reference, an array element reference, or a function reference.

A factor is a primary or a construct of the form:

*primary***primary*

A term is a factor or a construct of one of the forms:

term/*factor*
or
*term***term*

A signed term is a term immediately preceded by + or −.

A simple arithmetic expression is a term or two simple arithmetic expressions separated by a + or −.

An arithmetic expression is a simple arithmetic expression or a signed term or either of the preceding forms immediately followed by a + or − immediately followed by a simple arithmetic expression.

A primary of any type may be exponentiated by an integer primary, and the resultant factor is of the same type as that of the element being exponentiated. A real or double precision primary may be exponentiated by a real or double precision primary, and the resultant factor is of type real if both primaries are of type real and otherwise of type double precision. These are the only cases for which use of the exponentiation operator is defined.

By use of the arithmetic operators other than exponentiation, any admissible element may be combined with another admissible element of the same type, and the resultant element is of the same type. Further, an admissible real

element may be combined with an admissible double precision or complex element; the resultant element is of type double precision or complex, respectively.

6.2. Relational Expressions. A relational expression consists of two arithmetic expressions separated by a relational operator and will have the value true or false as the relation is true or false, respectively. One arithmetic expression may be of type real or double precision and the other of type real or double precision, or both arithmetic expressions may be of type integer. If a real expression and a double precision expression appear in a relational expression, the effect is the same as a similar relational expression. This similar expression contains a double precision zero as the right hand arithmetic expression and the difference of the two original expressions (in their original order) as the left. The relational operator is unchanged. The relational operators are:

Operator	*Representing*
.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to
.GT.	Greater than
.GE.	Greater than or equal to

6.3. Logical Expressions. A logical expression is formed with logical operators and logical elements and has the value true or false. The logical operators are:

Operator	*Representing*
.OR.	Logical disjunction
.AND.	Logical conjunction
.NOT.	Logical negation

The logical elements are logical primary, logical factor, logical term, and logical expression.

A logical primary is a logical expression enclosed in parentheses, a relational expression, a logical constant, a logical variable reference, a logical array element reference, or a logical function reference.

A logical factor is a logical primary or .NOT. followed by a logical primary.

A logical term is a logical factor or a construct of the form:

logical term .AND. logical term

A logical expression is a logical term or a construct of the form:

logical expression .OR. logical expression

6.4. Evaluation of Expressions. A part of an expression need be evaluated only if such action is necessary to establish the value of the expression. The rules for formation of expressions imply the binding strength of operators. It should be noted that the range of the subtraction operator is the term that immediately succeeds it. The evaluation may proceed according to any valid formation sequence.

When two elements are combined by an operator the order of evaluation of the elements is optional. If mathematical use of operators is associative, commutative, or both, full use of these facts may be made to revise orders of combination, provided only that integrity of parenthesized expressions is not violated. The value of an integer factor or term is the nearest integer whose magnitude does

not exceed the magnitude of the mathematical value represented by that factor or term.

Any use of an array element name requires the evaluation of its subscript. The evaluation of functions appearing in an expression may not validly alter the value of any other element within the expressions, assignment statement, or CALL statement in which the function reference appears. The type of the expression in which a function reference or subscript appears does not affect, nor is it affected by, the evaluation of the actual arguments or subscript.

No factor may be evaluated that requires a negative valued primary to be raised to a real or double precision exponent. No factor may be evaluated that requires raising a zero valued primary to a zero valued exponent.

No element may be evaluated whose value is not mathematically defined.

7. STATEMENTS

A statement may be classified as executable or nonexecutable. Executable statements specify actions; nonexecutable statements describe the characteristics and arrangements of data, editing information, statement functions, and classification of program units.

7.1. EXECUTABLE STATEMENTS. There are three types of executable statements:

1. Assignment statements.
2. Control statements.
3. Input/output statements.

7.1.1. *Assignment Statements.* There are three types of assignment statements:

1. Arithmetic assignment statement.
2. Logical assignment statement.
3. GO TO assignment statement.

7.1.1.1. *Arithmetic Assignment Statement.* An arithmetic assignment statement is of the form:

$$v = e$$

where v is a variable name or array element name of type other than logical and e is an arithmetic expression. Execution of this statement causes the evaluation of the expression e and the altering of v according to Table 1.

7.1.1.2. *Logical Assignment Statement.* A logical assignment statement is of the form

$$v = e$$

where v is a logical variable name or a logical array element name and e is a logical expression. Execution of this statement causes the logical expression to be evaluated and its value to be assigned to the logical entity.

7.1.1.3. GO TO *Assignment Statement.* A GO TO assignment statement is of the form:

ASSIGN k TO i

where k is a statement label and i is an integer variable name. After execution of such a statement, subsequent execution of any assigned GO TO statement (Section 7.1.2.1.2) using that integer variable will cause the statement identified by the assigned statement label to be executed next, provided there has been no intervening redefinition (9.2) of the variable. The statement label must refer to

an executable statement in the same program unit in which the ASSIGN statement appears.

Once having been mentioned in an ASSIGN statement, an integer variable may not be referenced in any statement other than an assigned GO TO statement until it has been redefined (Section 10.2.3).

TABLE 1. RULES FOR ASSIGNMENT OF *e* TO *v*

If v Type Is	*And e Type Is*	*The Assignment Rule Is**
Integer	Integer	Assign
Integer	Real	Fix & Assign
Integer	Double Precision	Fix & Assign
Integer	Complex	P
Real	Integer	Float & Assign
Real	Real	Assign
Real	Double Precision	DP Evaluate & Real Assign
Real	Complex	P
Double Precision	Integer	DP Float & Assign
Double Precision	Real	DP Evaluate & Assign
Double Precision	Double Precision	Assign
Double Precision	Complex	P
Complex	Integer	P
Complex	Real	P
Complex	Double Precision	P
Complex	Complex	Assign

* NOTES.

1. P means prohibited combination.
2. Assign means transmit the resulting value, without change, to the entity.
3. Real Assign means transmit to the entity as much precision of the most significant part of the resulting value as a real datum can contain.
4. DP Evaluate means evaluate the expression according to the rules of 6.1 (or any more precise rules) then DP Float.
5. Fix means truncate any fractional part of the result and transform that value to the form of an integer datum.
6. Float means transform the value to the form of a real datum.
7. DP Float means transform the value to the form of a double precision datum, retaining in the process as much of the precision of the value as a double precision datum can contain.

7.1.2. *Control Statements.* There are eight types of control statements:

1. GO TO statements.
2. Arithmetic IF statement.
3. Logical IF statement.
4. CALL statement.
5. RETURN statement.
6. CONTINUE statement.
7. Program control statements.
8. DO statement.

The statement labels used in a control statement must be associated with executable statements within the same program unit in which the control statement appears.

7.1.2.1. GO TO *Statements*. There are three types of GO TO statements:

1. Unconditional GO TO statement.
2. Assigned GO TO statement.
3. Computed GO TO statement.

7.1.2.1.1. *Unconditional* GO TO *Statement*. An unconditional GO TO statement is of the form:

GO TO k

where k is a statement label.

Execution of this statement causes the statement identified by the statement label to be executed next.

7.1.2.1.2. *Assigned* GO TO *Statement*. An assigned GO TO statement is of the form:

GO TO i, $(k_1, k_2, \ldots, k_n)$

where i is an integer variable reference, and the k's are statement labels.

At the time of execution of an assigned GO TO statement, the current value of i must have been assigned by the previous execution of an ASSIGN statement to be one of the statement labels in the parenthesized list, and such an execution causes the statement identified by that statement label to be executed next.

7.1.2.1.3. *Computed* GO TO *Statement*. A computed GO TO statement is of the form:

GO TO $(k_1, k_2, \ldots, k_n)$, i

where the k's are statement labels and i is an integer variable reference. See 10.2.8 and 10.3 for a discussion of requirements that apply to the use of a variable in a computed GO TO statement.

Execution of this statement causes the statement identified by the statement label k_j to be executed next, where j is the value of i at the time of the execution. This statement is defined only for values such that $1 \leqslant j \leqslant n$.

7.1.2.2. *Arithmetic* IF *Statement*. An arithmetic IF statement is of the form:

IF (e) k_1, k_2, k_3

where e is any arithmetic expression of type integer, real, or double precision, and the k's are statement labels.

The arithmetic IF is a three-way branch. Execution of this statement causes evaluation of the expression e following which the statement identified by the statement label k_1, k_2, or k_3 is executed next as the value of e is less than zero, zero, or greater than zero, respectively.

7.1.2.3. *Logical* IF *Statement*. A logical IF statement is of the form:

IF (e) S

where e is a logical expression and S is any executable statement except a DO statement or another logical IF statement. Upon execution of this statement, the logical expression e is evaluated. If the value of e is false, statement S is executed as though it were a CONTINUE statement. If the value of e is true, statement S is executed.

7.1.2.4. CALL *Statement*. A CALL statement is of one of the forms:

CALL s $(a_1, a_2, \ldots a_n)$
or
CALL s

where s is the name of a subroutine and the a's are actual arguments (8.4.2).

The inception of execution of a CALL statement references the designated subroutine. Return of control from the designated subroutine completes execution of the CALL statement.

7.1.2.5. RETURN *Statement.* A RETURN statement is of the form:

RETURN

A RETURN statement marks the logical end of a procedure subprogram and, thus, may only appear in a procedure subprogram.

Execution of this statement when it appears in a subroutine subprogram causes return of control to the current calling program unit.

Execution of this statement when it appears in a function subprogram causes return of control to the current calling program unit. At this time the value of the function (8.3.1) is made available.

7.1.2.6. CONTINUE *Statement.* A CONTINUE statement is of the form:

CONTINUE

Execution of this statement causes continuation of normal execution sequence.

7.1.2.7. *Program Control Statements.* There are two types of program control statements:

1. STOP statement.
2. PAUSE statement.

7.1.2.7.1. STOP *Statement.* A STOP statement is of one of the forms:

STOP n
or
STOP

where n is an octal digit string of length from one to five.

Execution of this statement causes termination of execution of the executable program.

7.1.2.7.2. PAUSE *Statement.* A PAUSE statement is of one of the forms:

PAUSE n
or
PAUSE

where n is an octal digit string of length from one to five.

The inception of execution of this statement causes a cessation of execution of this executable program. Execution must be resumable. At the time of cessation of execution the octal digit string is accessible. The decision to resume execution is not under control of the program; but if execution is resumed, execution of the PAUSE statement is completed.

7.1.2.8. DO *Statement.* A DO statement is of one of the forms:

DO n $i = m_1, m_2, m_3$
or
DO n $i = m_1, m_2$

where:

1. n is the statement of an executable statement. This statement, called the terminal statement of the associated DO, must physically follow and be in the same program unit as that DO statement. The terminal statement may not be a GO TO of any form, arithmetic IF, RETURN, STOP, PAUSE, or DO statement, nor a logical IF containing any of these forms.

2. i is an integer variable name; this variable is called the control variable.

3. m_1, called the initial parameter; m_2, called the terminal parameter; and m_3, called the incrementation parameter, are each either an integer constant or integer variable reference. If the second form of the DO statement is used so that m_3 is not explicitly stated, a value of one is implied for the incrementation parameter. At time of execution of the DO statement, m_1, m_2, and m_3 must be greater than zero.

Associated with each DO statement is a range that is defined to be those executable statements from and including the first executable statement following the DO, to and including the terminal statement associated with the DO. A special situation occurs when the range of a DO contains another DO statement. In this case, the range of the contained DO must be a subset of the range of the containing DO.

A completely nested nest is a set of DO statements and their ranges, and any DO statements contained within their ranges, such that the first occurring terminal statement of any of those DO statements physically follows the last occurring DO statement and the first occurring DO statement of the set is not in the range of any DO statement.

A DO statement is used to define a loop. The action succeeding execution of a Do statement is described by the following five steps:

1. The control variable is assigned the value represented by the initial parameter. This value must be less than or equal to the value represented by the terminal parameter.

2. The range of the DO is executed.

3. If control reaches the terminal statement, and after execution of the terminal statement, the control variable of the most recently executed DO statement associated with the terminal statement is incremented by the value represented by the associated incrementation parameter.

4. If the value of the control variable after incrementation is less than or equal to the value represented by the associated terminal parameter the action as described starting at step 2 is repeated with the understanding that the range in question is that of the DO, the control variable of which was most recently incremented. If the value of the control variable is greater than the value represented by its associated terminal parameter the DO is said to have been satisfied and the control variable becomes undefined.

5. At this point, if there were one or more other DO statements referring to the terminal statement in question the control variable of the next most recently executed DO statement is incremented by the value represented by its associated incrementation parameter and the action as described in step 4 is repeated until all DO statements referring to the particular termination statement are satisfied, at which time the first executable statement following the terminal statement is executed.

Upon exiting from the range of a DO by execution of a GO TO statement or an arithmetic IF statement, that is, other than by satisfying the DO, the control variable of the DO is defined and is equal to the most recent value attained as defined in the foregoing.

A DO is said to have an extended range if both of the following conditions apply:

1. There exists a GO TO statement or arithmetic IF statement within the range of the innermost DO of a completely nested nest that can cause control to pass out of that nest.

2. There exists a GO TO statement or arithmetic IF statement not within the

nest that, in the collection of all possible sequences of execution in the particular program unit could be executed after a statement of the type described in (1), and the execution of which could cause control to return into the range of the innermost DO of the completely nested nest.

If both of these conditions apply, the extended range is defined to be the set of all executable statements that may be executed between all pairs of control statements, the first of which satisfies the condition of (1) and the second of (2). The first of the pair is not included in the extended range; the second is. A GO TO statement or an arithmetic IF statement may not cause control to pass into the range of a DO unless it is being executed as part of the extended range of that particular DO. Further, the extended range of a DO may not contain a Do that has an extended range. When a procedure reference occurs in the range of a DO, the actions of that procedure are considered to be temporarily within that range, i.e. during the execution of that reference.

The control variable, initial parameter, terminal parameter, and incrementation parameter of a DO may not be redefined during the execution of the range or extended range of that DO.

If a statement is the terminal statement of more than one DO statement the statement label of that terminal statement may not be used in any GO TO or arithmetic IF statement that occurs anywhere but in the range of the most deeply contained DO with that terminal statement.

7.1.3. *Input/Output Statements.* There are two types of input/output statements:

1. READ and WRITE statements.
2. Auxiliary Input/Output statements.

The first type consists of the statements that cause transfer of records of sequential files to and from internal storage, respectively. The second type consists of the BACKSPACE and REWIND statements that provide for positioning of such an external file, and ENDFILE, which provides for demarcation of such an external file.

In the following descriptions u and f identify input/output units and format specifications, respectively. An input/output unit is identified by an integer value and u may be either an integer constant or an integer variable reference whose value then identifies the unit. The format specification is described in Section 7.2.3. Either the statement label of a FORMAT statement or an array name may be represented by f. If a statement label, the identified statement must appear in the same program unit as the input/output statement. If an array name, it must conform to the specifications in 7.2.3.10.

A particular unit has a single sequential file associated with it. The most general case of such a unit has the following properties:

1. If the unit contains one or more records, those records exist as a totally ordered set.

2. There exists a unique position of the unit called its initial point. If a unit contains no records, that unit is positioned at its initial point. If the unit is at its initial point and contains records, the first record of the unit is defined as the next record.

3. If a unit is not positioned at its initial point, there exists a unique preceding record associated with that position. The least of any records in the ordering described by (1) following this preceding record is defined as the next record of that position.

4. Upon completion of execution of a WRITE or ENDFILE statement, there exist no records following the records created by that statement.

5. When the next record is transmitted, the position of the unit is changed so that this next record becomes the preceding record.

If a unit does not provide for some of the properties given in the foregoing, certain statements that will be defined may not refer to that unit. The use of such a statement is not defined for that unit.

7.1.3.1. READ *and* WRITE *Statements.* The Read and WRITE statements specify transfer of information. Each such statement may include a list of the names of variables, arrays, and array elements. The named elements are assigned values on input and have their values transferred on output.

Records may be formatted or unformatted. A formatted record consists of a string of the characters that are permissible in Hollerith constants (5.1.1.6). The transfer of such a record requires that a format specification be referenced to supply the necessary positioning and conversion specifications (7.2.3). The number of records transferred by the execution of a formatted READ or WRITE is dependent upon the list and referenced format specification (7.2.3.4). An unformatted record consists of a string of values. When an unformatted or formatted READ statement is executed, the required records on the identified unit must be, respectively, unformatted or formatted records.

7.1.3.1.1. *Input/Output Lists.* The input list specifies the names of the variables and array elements to which values are assigned on input. The output list specifies the references to variables and array elements whose values are transmitted. The input and output lists are of the same form.

Lists are formed in the following manner. A simple list is a variable name, an array element name, or an array name, or two simple lists separated by a comma.

A list is a simple list, a simple list enclosed in parentheses, a DO-implied list, or two lists separated by a comma.

A DO-implied list is a list followed by a comma and a DO-implied specification, all enclosed in parentheses.

A DO-implied specification is of one of the forms:

$$i = m_1, m_2, m_3$$
or
$$i = m_1, m_2$$

The elements i, m_1, m_2, and m_3 are as defined for the DO statement (7.1.2.8). The range of DO-implied specification is the list of the DO-implied list and, for input lists, i, m_1, m_2, and m_3 may appear, within that range, only in subscripts.

A variable name or array element name specifies itself. An array name specifies all of the array element names defined by the array declarator, and they are specified in the order given by the array element successor function (7.2.1.1.1).

The elements of a list are specified in the order of their occurrence from left to right. The elements of a list in a DO-implied list are specified for each cycle of the implied DO.

7.1.3.1.2. *Formatted* READ. A formatted READ statement is of one of the forms:

READ $(u, f)k$
or
READ (u, f)

where k is a list.

Execution of this statement causes the input of the next records from the unit identified by u. The information is scanned and converted as specified by the

format specification identified by *f*. The resulting values are assigned to the elements specified by the list. See, however, 7.2.3.4.

7.1.3.1.3. *Formatted* WRITE. A formatted WRITE statement is of one of the forms:

WRITE (*u*, *f*) *k*

or

WRITE (*u*, *f*)

where *k* is a list.

Execution of this statement creates the next records on the unit identified by *u*. The list specifies a sequence of values. These are converted and positioned as specified by the format specification identified by *f*. See, however, 7.2.3.4.

7.1.3.1.4. *Unformatted* READ. An unformatted READ statement is of one of the forms:

READ (*u*) *k*

or

READ (*u*)

where *k* is a list.

Execution of this statement causes the input of the next record from the unit identified by *u*, and if there is a list these values are assigned to the sequence of elements specified by the list. The sequence of values required by the list may not exceed the sequence of values from the unformatted record.

7.1.3.1.5. *Unformatted* WRITE. An unformatted WRITE statement is of the form:

WRITE (*u*) *k*

where *k* is a list.

Execution of this statement creates the next record on the unit identified by *u* of the sequence of values specified by the list.

7.1.3.2. *Auxiliary Input/Output Statements.* There are three types of auxiliary input/output statements:

1. REWIND statement.
2. BACKSPACE statement.
3. ENDFILE statement.

7.1.3.2.1. REWIND *Statement.* A REWIND statement is of the form:

REWIND *u*

Execution of this statement causes the unit identified by *u* to be positioned at its initial point.

7.1.3.2.2. BACKSPACE *Statement.* A BACKSPACE statement is of the form:

BACKSPACE *u*

If the unit identified by *u* is positioned at its initial point, execution of this statement has no effect. Otherwise, the execution of this statement results in the positioning of the unit identified by *u* so that what had been the preceding record prior to that execution becomes the next record.

7.1.3.2.3. ENDFILE *Statement.* An ENDFILE statement is of the form:

ENDFILE *u*

Execution of this statement causes the recording of an endfile record on the unit identified by *u*. The endfile record is an unique record signifying a demarcation of a sequential file. Action is undefined when an endfile record is encountered during execution of a READ statement.

7.1.3.3. *Printing of Formatted Records.* When formatted records are prepared for printing the first character of the record is not printed.

The first character of such a record determines vertical spacing as follows:

Character	*Vertical Spacing Before Printing*
Blank	One line
0	Two lines
1	To first line of next page
+	No advance

7.2. NONEXECUTABLE STATEMENTS. There are five types of nonexecutable statements:

1. Specification statements.
2. Data initialization statement.
3. FORMAT statement.
4. Function defining statements.
5. Subprogram statements.

See 10.1.2 for a discussion of restrictions on appearances of symbolic names in such statements.

The function defining statements and subprogram statements are discussed in Section 8.

7.2.1. *Specification Statements.* There are five types of specification statements:

1. DIMENSION statement.
2. COMMON statement.
3. EQUIVALENCE statement.
4. EXTERNAL statement.
5. Type-statements.

7.2.1.1. *Array-Declarator.* An array declarator specifies an array used in a program unit.

The array declarator indicates the symbolic name, the number of dimensions (one, two, or three), and the size of each of the dimensions. The array declarator statement may be a type-statement, DIMENSION, or COMMON statement.

An array declarator has the form:

$$v\,(i)$$

where:

1. v, called the declarator name, is a symbolic name.
2. (i), called the declarator subscript, is composed of 1, 2, or 3 expressions, each of which may be an integer constant or an integer variable name. Each expression is separated by a comma from its successor if there are more than one of them. In the case where i contains no integer variable, i is called the constant declarator subscript.

The appearance of a declarator subscript in a declarator statement serves to inform the processor that the declarator name is an array name. The number of subscript expressions specified for the array indicates its dimensionality. The magnitude of the values given for the subscript expressions indicates the maximum value that the subscript may attain in any array element name.

No array element name may contain a subscript that, during execution of the executable program, assumes a value less than one or larger than the maximum length specified in the array declarator.

7.2.1.1.1. *Array Element Successor Function and Value of a Subscript.* For a given dimensionality, subscript declarator, and subscript, the value of a subscript pointing to an array element and the maximum value a subscript may attain is indicated in Table 2. A subscript expression must be greater than zero.

The value of the array element successor function is obtained by adding one to the entry in the subscript value column. Any array element whose subscript has this value is the successor to the original element. The last element of the array is the one whose subscript value is the maximum subscript value and has no successor element.

TABLE 2. VALUE OF A SUBSCRIPT

Dimensionality	*Subscript Declarator*	*Subscript*	*Subscript Value*	*Maximum Subscript Value*
1	(A)	(a)	a	A
2	(A, B)	(a, b)	$a + A \cdot (b - 1)$	$A \cdot B$
3	(A, B, C)	(a, b, c)	$a + A \cdot (b - 1) + A \cdot B \cdot (c - 1)$	$A \cdot B \cdot C$

NOTES. 1. a, b, and c are subscript expressions.
2. A, B, and C are dimensions.

7.2.1.1.2. *Adjustable Dimension.* If any of the entries in a declarator subscript is an integer variable name the array is called an adjustable array, and the variable names are called adjustable dimensions. Such an array may appear only in a subprogram. The dummy argument list of the subprogram must contain the array name and the integer variable names that represent the adjustable dimensions. The values of the actual arguments that represent array dimensions in the argument list of the reference must be defined (10.2) prior to calling the subprogram and may not be redefined or undefined during execution of the subprogram. The maximum size of the actual array may not be exceeded. For every array appearing in an executable program (9.1.6.) there must be at least one constant array declarator associated through subprogram references.

In a subprogram a symbolic name that appears in a COMMON statement may not identify an adjustable array.

7.2.1.2. DIMENSION *Statement.* A DIMENSION statement is of the form:

$$\text{DIMENSION } v_1(i_1), v_2(i_2), \ldots, v_n(i_n)$$

where each $v(i)$ is an array declarator.

7.2.1.3. COMMON *Statement.* A COMMON statement is of the form:

$$\text{COMMON } / x_1 / a_1 / \ldots / x_n / a_n$$

where each a is a nonempty list of variable names, array names, or array declarators (no dummy arguments are permitted) and each x is a symbolic name or is empty. If x_1 is empty the first two slashes are optional. Each x is a block name, a name that bears no relationship to any variable or array having the same name. This holds true for any such variable or array in the same or any other program unit. See 10.1.1 for a discussion of restrictions on uses of block names.

In any given COMMON statement the entities occurring between block name x and the next block name (or the end of the statement if no block name follows) are declared to be in common block x. All entities from the beginning of the statement until the appearance of a block name, or all entities in the statement

if no block name appears, are declared to be in blank or unlabelled common. Alternatively, the appearance of two slashes with no block name between them declares the entities that follow to be in blank common.

A given common block name may occur more than once in a COMMON statement or in a program unit. The processor will string together in a given common block all entities so assigned in the order of their appearance (10.1.2). The first element of an array will follow the immediately preceding entity, if one exists, and the last element of an array will immediately precede the next entity, if one exists.

The size of a common block in a program unit is the sum of the storage required for the elements introduced through COMMON and EQUIVALENCE statements. The sizes of labelled common blocks with the same label in the program units that comprise an executable program must be the same. The sizes of blank common in the various program units that are to be executed together need not be the same. Size is measured in terms of storage units (7.2.1.3.1).

7.2.1.3.1. *Correspondence of Common Blocks.* If all of the program units of an executable program that contain any definition of a common block of a particular name define that block such that:

1. There is identity in type for all entities defined in the corresponding position from the beginning of that block.

2. If the block is labelled and the same number of entities is defined for the block, then the values in the corresponding positions (counted by the number of preceding storage units) are the same quantity in the executable program.

A double precision or a complex entity is counted as two logically consecutive storage units; a logical, real, or integer entity, as one storage unit.

Then for common blocks with the same number of storage units or blank common:

1. In all program units which have defined the identical type to a given position (counted by the number of preceding storage units) references to that position refer to the same quantity.

2. A correct reference is made to a particular position assuming a given type if the most recent value assignment to that position was of the same type.

7.2.1.4. EQUIVALENCE *Statement.* An EQUIVALENCE statement is of the form:

$$\text{EQUIVALENCE } (k_1), (k_2), \ldots, (k_n)$$

in which each k is a list of the form:

$$a_1, a_2, \ldots, a_m.$$

Each a is either a variable name or an array element name (not a dummy argument), the subscript of which contains only constants, and m is greater than or equal to two. The number of subscript expressions of an array element name must correspond in number to the dimensionality of the array declarator or must be one (the array element successor function defines a relation by which an array can be made equivalent to a one dimensional array of the same length).

The EQUIVALENCE statement is used to permit the sharing of storage by two or more entities. Each element in a given list is assigned the same storage (or part of the same storage) by the processor. The EQUIVALENCE statement should not be used to equate mathematically two or more entities. If a two-storage unit entity is equivalenced to a one-storage unit entity the latter will share space with the first storage unit of the former.

The assignment of storage to variables and arrays declared directly in a COMMON statement is determined solely by consideration of their type and the

COMMON and array declarator statements. Entities so declared are always assigned unique storage, contiguous in the order declared in the COMMON statement.

The effect of an EQUIVALENCE statement upon common assignment may be the lengthening of a common block; the only such lengthening permitted is that which extends a common block beyond the last assignment for that block made directly by a COMMON statement.

When two variables or array elements share storage because of the effects of EQUIVALENCE statements the symbolic names of the variables or arrays in question may not both appear in COMMON statements in the same program unit.

Information contained in 7.2.1.1.1, 7.2.1.3.1, and the present section suffices to describe the possibilities of additional cases of sharing of storage between array elements and entities of common blocks. It is incorrect to cause either directly or indirectly a single storage unit to contain more than one element of the same array.

7.2.1.5. EXTERNAL *Statement.* An EXTERNAL statement is of the form:

$$\text{EXTERNAL } v_1, v_2, \ldots, v_n$$

where each v is an external procedure name.

Appearance of a name in an EXTERNAL statement declares that name to be an external procedure name. If an external procedure name is used as an argument to another external procedure it must appear in an EXTERNAL statement in the program unit in which it is so used.

7.2.1.6. *Type-statements.* A type-statement is of the form:

$$t\ v_1, v_2, \ldots, v_n$$

where t is INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or LOGICAL, and each v is a variable name, an array name, a function name, or an array declarator.

A type-statement is used to override or confirm the implicit typing, to declare entities to be of type double precision, complex, or logical, and may supply dimension information.

The appearance of a symbolic name in a type-statement serves to inform the processor that it is of the specified data type for all appearances in the program unit.

7.2.2. *Data Initialization Statement.* A data initialization statement is of the form:

$$\text{DATA } k_1 / d_1 / , k_2 / d_2 / , \ldots , k_n / d_n /$$

where:

1. Each k is a list containing names of variables and array elements.
2. Each d is a list of constants and optionally signed constants, any of which may be preceded by $j*$.
3. j is an integer constant.

If a list contains more than one entry the entries are separated by commas.

Dummy arguments may not appear in the list k. Any subscript expression must be an integer constant.

When the form $j*$ appears before a constant it indicates that the constant is to be specified j times. A Hollerith constant may appear in the list d.

A data initialization statement is used to define initial values of variables or array elements. There must be a one-to-one correspondence between the list-

specified items and the constants. By this correspondence, the initial value is established.

An initially defined variable or array element may not be in blank common. A variable or array element in a labelled common block may be initially defined only in a block data subprogram.

7.2.3. FORMAT *Statement.* FORMAT statements are used in conjunction with the input/output of formatted records to provide conversion and editing information between the internal representation and the external character strings.

A FORMAT statement is of the form:

FORMAT $(q_1t_1z_1t_2z_2 \ldots t_nz_nq_2)$

where:

1. $(q_1t_1z_1t_2z_2 \ldots t_nz_nq_2)$ is the format specification.
2. Each q is a series of slashes or is empty.
3. Each t is a field descriptor or group of field descriptors.
4. Each z is a field separator.
5. n may be zero.

A FORMAT statement must be labelled.

7.2.3.1. *Field Descriptors.* The format field descriptors are of the forms:

srF$w.d$
srE$w.d$
srG$w.d$
srD$w.d$
rIw
rLw
rAw
nH$h_1h_2 \ldots h_n$
nX

where:

1. The letters F, E, G, D, I, L, A, H, and X indicate the manner of conversion and editing between the internal and external representations and are called the conversion codes.
2. w and n are nonzero integer constants representing the width of the field in the external character string.
3. d is an integer constant representing the number of digits in the fractional part of the external character string (except for G conversion code).
4. r, the repeat count, is an optional nonzero integer constant indicating the number of times to repeat the succeeding basic field descriptor.
5. s is optional and represents a scale factor designator.
6. Each h is one of the characters capable of representation by the processor.

For all descriptors, the field width must be specified. For descriptors of the form $w.d$, the d must be specified, even if it is zero. Further, w must be greater than or equal to d.

The phrase *basic field descriptor* will be used to signify the field descriptor unmodified by s or r.

The internal representation of external fields will correspond to the internal representation of the corresponding type constants (4.2 and 5.1.1).

7.2.3.2. *Field Separators.* The format field separators are the slash and the comma. A series of slashes is also a field separator. The field descriptors or groups of field descriptors are separated by a field separator.

The slash is used not only to separate field descriptors but also to specify demarcation of formatted records. A formatted record is a string of characters. The lengths of the strings for a given external medium are dependent upon both the processor and the external medium.

The processing of the number of characters that can be contained in a record by an external medium does not of itself cause the introduction or inception of processing of the next record.

7.2.3.3. *Repeat Specifications.* Repetition of the field descriptors (except nH and nX) is accomplished by using the repeat count. If the input/output list warrants, the specified conversion will be interpreted repetitively up to the specified number of times.

Repetition of a group of field descriptors or field separators is accomplished by enclosing them within parentheses and optionally preceding the left parenthesis with an integer constant called the group repeat count indicating the number of times to interpret the enclosed grouping. If no group repeat count is specified a group repeat count of one is assumed. This form of grouping is called a basic group.

A further grouping may be formed by enclosing field descriptors, field separators, or basic groups within parentheses. Again, a group repeat count may be specified. The parentheses enclosing the format specification are not considered as group delineating parentheses.

7.2.3.4. *Format Control Interaction with an Input/Output List.* The inception of execution of a formatted READ or formatted WRITE statement initiates format control. Each action of format control depends on information jointly provided respectively by the next element of the input/output list, if one exists, and the next field descriptor obtained from the format specification. If there is an input/output list at least one field descriptor other than nH or nX must exist.

When a READ statement is executed under format control one record is read when the format control is initiated, and thereafter additional records are read only as the format specification demands. Such action may not require more characters of a record than it contains.

When a WRITE statement is executed under format control, writing of a record occurs each time the format specification demands that a new record be started. Termination of format control causes writing of the current record.

Except for the effects of repeat counts, the format specification is interpreted from left to right.

To each I, F, E, G, D, A, or L basic descriptor interpreted in a format specification, there corresponds one element specified by the input/output list, except that a complex element requires the interpretation of two F, E, or G basic descriptors. To each H or X basic descriptor there is no corresponding element specified by the input/output list, and the format control communicates information directly with the record. Whenever a slash is encountered, the format specification demands that a new record start or the preceding record terminate. During a READ operation, any unprocessed characters of the current record will be skipped at the time of termination of format control or when a slash is encountered.

Whenever the format control encounters an I, F, E, G, D, A, or L basic descriptor in a format specification, it determines if there is a corresponding element specified by the input/output list. If there is such an element, it transmits

appropriately converted information between the element and the record and proceeds. If there is no corresponding element, the format control terminates.

If, however, the format control proceeds to the last outer right parenthesis of the format specification, a test is made to determine if another list element is specified. If not, control terminates. However, if another list element is specified the format control demands a new record start and control reverts to that group repeat specification terminated by the last preceding right parenthesis, or if none exists, then to the first left parenthesis of the format specification. Note, this action of itself has no effect on the scale factor.

7.2.3.5. *Scale Factor.* A scale factor designator is defined for use with the F, E, G, and D conversions and is of the form:

$$n\text{P}$$

where n, the scale factor, is an integer constant or minus followed by an integer constant.

When the format control is initiated a scale factor of zero is established. Once a scale factor has been established, it applies to all subsequently interpreted F, E, G, and D field descriptors, until another scale factor is encountered, and then that scale factor is established.

7.2.3.5.1. *Scale Factor Effects.* The scale factor n affects the appropriate conversions in the following manner:

1. For F, E, G, and D input conversions (provided no exponent exists in the external field) and F output conversions, the scale factor effect is as follows:

   Externally represented number equals internally represented number times the quantity ten raised to the nth power.

2. For F, E, G, and D input, the scale factor has no effect if there is an exponent in the external field.

3. For E and D output, the basic real constant part of the output quantity is multiplied by 10^n and the exponent is reduced by n.

4. For G output, the effect of the scale factor is suspended unless the magnitude of the datum to be converted is outside the range that permits the effective use of F conversion. If the effective use of E conversion is required, the scale factor has the same effect as with E output.

7.2.3.6. *Numeric Conversions.* The numeric field descriptors I, F, E, G, and D are used to specify input/output of integer, real, double precision, and complex data.

1. With all numeric input conversions, leading blanks are not significant and other blanks are zero. Plus signs may be omitted. A field of all blanks is considered to be zero.

2. With the F, E, G, and D input conversions, a decimal point appearing in the input field overrides the decimal point specification supplied by the field descriptor.

3. With all output conversions, the output field is right justified. If the number of characters produced by the conversion is smaller than the field width leading blanks will be inserted in the output field.

4. With all output conversions, the external representation of a negative value must be signed; a positive value may be signed.

5. The number of characters produced by an output conversion must not exceed the field width.

7.2.3.6.1. *Integer Conversion.* The numeric field descriptor Iw indicates that the external field occupies w positions as an integer. The value of the list item appears, or is to appear, internally as an integer datum.

In the external input field, the character string must be in the form of an integer constant or signed integer constant (5.1.1.1), except for the interpretation of blanks (7.2.3.6).

The external output field consists of blanks, if necessary, followed by a minus if the value of the internal datum is negative, or an optional plus otherwise, followed by the magnitude of the internal value converted to an integer constant.

7.2.3.6.2. *Real Conversions.* There are three conversions available for use with real data: F, E, and G.

The numeric field descriptor F$w.d$ indicates that the external field occupies w positions, the fractional part of which consists of d digits. The value of the list item appears, or is to appear, internally as a real datum.

The basic form of the external input field consists of an optional sign, followed by a string of digits optionally containing a decimal point. The basic form may be followed by an exponent of one of the following forms:

1. Signed integer constant.
2. E followed by an integer constant.
3. E followed by a signed integer constant.
4. D followed by an integer constant.
5. D followed by a signed integer constant.

An exponent containing D is equivalent to an exponent containing E.

The external output field consists of blanks, if necessary, followed by a minus if the internal value is negative, or an optional plus otherwise, followed by string of digits containing a decimal point representing the magnitude of the internal value, as modified by the established scale factor, rounded to d fractional digits.

The numeric field descriptor E$w.d$ indicates that the external field occupies w positions, the fractional part of which consists of d digits. The value of the list item appears, or is to appear, internally as a real datum.

The form of the external input field is the same as for the F conversion.

The standard form of the external output field for a scale factor of zero is*

* ξ signifies no character position or minus in that position.

$$\xi 0.x_1 \ldots x_d Y$$

where:

1. $x_1 \ldots x_d$ are the d most significant rounded digits of the value of the data to be output.
2. Y is of one of the forms:

$$\mathrm{E} \pm y_1 y_2$$
or
$$\pm y_1 y_2 y_3$$

and has the significance of a decimal exponent (an alternative for the plus in the first of these forms is the character blank).

3. The digit 0 in the aforementioned standard form may optionally be replaced by no character position.
4. Each y is a digit.

The scale factor n controls the decimal normalization between the number part and the exponent part such that:

1. If $n \leqslant 0$ there will be exactly $-n$ leading zeros and $d+n$ significant digits after the decimal point.

2. If $n > 0$ there will be exactly n significant digits to the left of the decimal point and $d-n+1$ to the right of the decimal point.

The numeric field descriptor G$w.d$ indicates that the external field occupies w positions with d significant digits. The value of the list item appears, or is to appear, internally as a real datum.

Input processing is the same as for the F conversion.

The method of representation in the external output string is a function of the magnitude of the real datum being converted. Let N be the magnitude of the internal datum. The following tabulation exhibits a correspondence between N and the equivalent method of conversion that will be effected:

Magnitude of Datum	*Equivalent Conversion Effected*
$0.1 \leqslant N < 1$	F$(w-4).d$, 4X
$1 \leqslant N < 10$	F$(w-4).(d-1)$, 4X
⋮	⋮
$10^{d-2} \leqslant N < 10^{d-1}$	F$(w-4).1$, 4X
$10^{d-1} \leqslant N < 10^{d}$	F$(w-4).0$, 4X
Otherwise	sE$w.d$

Note that the effect of the scale factor is suspended unless the magnitude of the datum to be converted is outside of the range that permits effective use of F conversion.

7.2.3.6.3. *Double Precision Conversion.* The numeric field descriptor D$w.d$ indicates that the external field occupies w positions, the fractional part of which consists of d digits. The value of the list item appears, or is to appear, internally as a double precision datum.

The basic form of the external input field is the same as for real conversions.

The external output field is the same as for the E conversion, except that the character D may replace the character E in the exponent.

7.2.3.6.4. *Complex Conversion.* Since a complex datum consists of a pair of separate real data, the conversion is specified by two successively interpreted real field descriptors. The first of these supplies the real part. The second supplies the imaginary part.

7.2.3.7. *Logical Conversion.* The logical field descriptor Lw indicates that the external field occupies w positions as a string of information as defined below. The list item appears, or is to appear, internally as a logical datum.

The external input field must consist of optional blanks followed by a T or F followed by optional characters, for true and false, respectively.

The external output field consists of $w-1$ blanks followed by a T or F as the value of the internal datum is true or false, respectively.

7.2.3.8. *Hollerith Field Descriptor.* Hollerith information may be transmitted by means of two field descriptors, nH and Aw :

1. The nH descriptor causes Hollerith information to be read into, or written from, the n characters (including blanks) following the nH descriptor in the format specification itself.

2. The Aw descriptor causes w Hollerith characters to be read into, or written from, a specified list element.

Let g be the number of characters representable in a single storage unit (7.2.1.3.1). If the field width specified for A input is greater than or equal to g

the rightmost g characters will be taken from the external input field. If the field width is less than g the w characters will appear left justified with $w-g$ trailing blanks in the internal representation.

If the field width specified for A output is greater than g the external output field will consist of $w-g$ blanks, followed by the g characters from the internal representation. If the field width is less than or equal to g the external output field will consist of the leftmost w characters from the internal representation.

7.2.3.9. *Blank Field Descriptor.* The field descriptor for blanks is nX. On input, n characters of the external input record are skipped. On output, n blanks are inserted in the external output record.

7.2.3.10. *Format Specification in Arrays.* Any of the formatted input/output statements may contain an array name in place of the reference to a FORMAT statement label. At the time an array is referenced in such a manner the first part of the information contained in the array, taken in the natural order, must constitute a valid format specification. There is no requirement on the information contained in the array following the right parenthesis that ends the format specification.

The format specification which is to be inserted in the array has the same form as that defined for a FORMAT statement; that is, begins with a left parenthesis and ends with a right parenthesis. An nH field descriptor may not be part of a format specification within an array.

8. PROCEDURES AND SUBPROGRAMS

There are four categories of procedures: statement functions, intrinsic functions, external functions, and external subroutines. The first three categories are referred to collectively as functions or function procedures; the last as subroutines or subroutine procedures. There are two categories of subprograms: procedure subprograms and specification subprograms. Function subprograms and subroutine subprograms are classified as procedure subprograms. Block data subprograms are classified as specification subprograms. Type rules for function procedures are given in 5.3.

8.1. Statement Functions. A statement function is defined internally to the program unit in which it is referenced. It is defined by a single statement similar in form to an arithmetic or logical assignment statement.

In a given program unit all statement function definitions must precede the first executable statement of the program unit and must follow the specification statements, if any. The name of a statement function must not appear in an EXTERNAL statement, nor as a variable name or an array name in the same program unit.

8.1.1. *Defining Statement Functions.* A statement function is defined by a statement of the form:

$$f(a_1, a_2, \ldots, a_n) = e$$

where f is the function name, e is an expression, and the relationship between f and e must conform to the assignment rules in 7.1.1.1 and 7.1.1.2. The a's are distinct variable names, called the *dummy arguments* of the function. Since these are dummy arguments, their names, which serve only to indicate type, number, and order of arguments, may be the same as variable names of the same type appearing elsewhere in the program unit.

Aside from the dummy arguments, the expression *e* may only contain:

1. Non-Hollerith constants.
2. Variable references.
3. Intrinsic function references.
4. References to previously defined statement functions.
5. External function references.

8.1.2. *Referencing Statement Functions.* A statement function is referenced by using its reference (5.2) as a primary in an arithmetic or logical expression. The actual arguments, which constitute the argument list, must agree in order, number, and type with the corresponding dummy arguments. An actual argument in a statement function reference may be any expression of the same type as the corresponding dummy argument.

Execution of a statement function reference results in an association (10.2.2) of actual argument values with the corresponding dummy arguments in the expression of the function definition, and an evaluation of the expression. Following this, the resultant value is made available to the expression that contained the function reference.

8.2. INTRINSIC FUNCTIONS AND THEIR REFERENCE. The symbolic names of the intrinsic functions (see Table 3) are predefined to the processor and have a special meaning and type if the name satisfies the conditions of 10.1.7.

An intrinsic function is referenced by using its reference as a primary in an arithmetic or logical expression. The actual arguments, which constitute the argument list, must agree in type, number, and order with the specification in Table 3 and may be any expression of the specified type. The intrinsic functions AMOD, MOD, SIGN, ISIGN, and DSIGN are not defined when the value of the second argument is zero.

Execution of an intrinsic function reference results in the actions specified in Table 3 based on the values of the actual arguments. Following this, the resultant value is made available to the expression that contained the function reference.

8.3. EXTERNAL FUNCTIONS. An external function is defined externally to the program unit that references it. An external function defined by FORTRAN statements headed by a FUNCTION statement is called a function subprogram.

8.3.1. *Defining Function Subprograms.* A FUNCTION statement is of the form:

$$t \text{ FUNCTION } f\,(a_1, a_2, \ldots, a_n)$$

where:

1. t is either INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or LOGICAL, or is empty.
2. f is the symbolic name of the function to be defined.
3. The a's, called the dummy arguments, are each either a variable name, an array name, or an external procedure name.

Function subprograms are constructed as specified in 9.1.3 with the following restrictions:

1. The symbolic name of the function must also appear as a variable name in the defining subprogram. During every execution of the subprogram this variable must be defined and, once defined, may be referenced or redefined. The value of the variable at the time of execution of any RETURN statement in this subprogram is called the value of the function.
2. The symbolic name of the function must not appear in any nonexecutable

TABLE 3. INTRINSIC FUNCTIONS

Intrinsic Function	*Definition*	*Number of Arguments*	*Symbolic Name*	*Type of:*	
				Argument	*Function*
Absolute Value	$\lvert a \rvert$	1	ABS IABS DABS	Real Integer Double	Real Integer Double
Truncation	Sign of a times largest integer $\leqslant \lvert a \rvert$	1	AINT INT IDINT	Real Real Double	Real Integer Integer
Remaindering* (see note below)	$a_1 \pmod{a_2}$	2	AMOD MOD	Real Integer	Real Integer
Choosing Largest Value	Max $(a_1, a_2, \ldots)$	$\geqslant 2$	AMAX0 AMAX1 MAX0 MAX1 DMAX1	Integer Real Integer Real Double	Real Real Integer Integer Double
Choosing Smallest Value	Min $(a_1, a_2, \ldots)$	$\geqslant 2$	AMIN0 AMIN1 MIN0 MIN1 DMIN1	Integer Real Integer Real Double	Real Real Integer Integer Double
Float	Conversion from integer to real	1	FLOAT	Integer	Real
Fix	Conversion from real to integer	1	IFIX	Real	Integer
Transfer of Sign	Sign of a_2 times $\lvert a_1 \rvert$	2	SIGN ISIGN DSIGN	Real Integer Double	Real Integer Double
Positive Difference	$a_1 - \text{Min}\,(a_1, a_2)$	2	DIM IDIM	Real Integer	Real Integer
Obtain Most Significant Part of Double Precision Argument		1	SNGL	Double	Real
Obtain Real Part of Complex Argument		1	REAL	Complex	Real
Obtain Imaginary Part of Complex Argument		1	AIMAG	Complex	Real
Express Single Precision Argument in Double Precision Form		1	DBLE	Real	Double
Express Two Real Arguments in Complex Form	$a_1 + a_2\sqrt{-1}$	2	CMPLX	Real	Complex
Obtain Conjugate of a Complex Argument		1	CONJG	Complex	Complex

* The function MOD or AMOD (a_1, a_2) is defined as $a_1 - [a_1/a_2]a_2$, where $[x]$ is the integer whose magnitude does not exceed the magnitude of x and whose sign is the same as x.

statement in this program unit, except as the symbolic name of the function in the FUNCTION statement.

3. The symbolic names of the dummy arguments may not appear in an EQUIVALENCE, COMMON, or DATA statement in the function subprogram.

4. The function subprogram may define or redefine one or more of its arguments so as to effectively return results in addition to the value of the function.

5. The function subprogram may contain any statements except BLOCK DATA, SUBROUTINE, another FUNCTION statement, or any statement that directly or indirectly references the function being defined.

6. The function subprogram must contain at least one RETURN statement.

8.3.2. *Referencing External Functions.* An external function is referenced by using its reference (5.2) as a primary in an arithmetic or logical expression. The actual arguments, which constitute the argument list, must agree in order, number, and type with the corresponding dummy arguments in the defining program unit. An actual argument in an external function reference may be one of the following:

1. A variable name.
2. An array element name.
3. An array name.
4. Any other expression.
5. The name of an external procedure.

If an actual argument is an external function name or a subroutine name, then the corresponding dummy argument must be used as an external function name or a subroutine name, respectively.

If an actual argument corresponds to a dummy argument that is defined or redefined in the referenced subprogram the actual argument must be a variable name, an array element name, or an array name. Execution of an external function reference as described in the foregoing results in an association (10.2.2) of actual arguments with all appearances of dummy arguments in executable statements, function definition statements, and as adjustable dimensions in the defining subprogram. If the actual argument is as specified in item (4) in the foregoing this association is by value rather than by name. Following these associations, execution of the first executable statement of the defining subprogram is undertaken. An actual argument which is an array element name containing variables in the subscript could in every case be replaced by the same argument with a constant subscript containing the same values as would be derived by computing the variable subscript just before the association of arguments takes place.

If a dummy argument of an external function is an array name the corresponding actual argument must be an array name or array element name (10.1.3).

If a function reference causes a dummy argument in the referenced function to become associated with another dummy argument in the same function or with an entity in common a definition of either within the function is prohibited.

Unless it is a dummy argument, an external function is also referenced (in that it must be defined) by the appearance of its symbolic name in an EXTERNAL statement.

8.3.3. *Basic External Functions.* FORTRAN processors must supply the external functions listed in Table 4. Referencing of these functions is accomplished as described in (8.3.2). Arguments for which the result of these functions is not mathematically defined or is of type other than that specified are improper.

8.4. SUBROUTINE. An external subroutine is defined externally to the program unit that references it. An external subroutine defined by FORTRAN statements headed by a SUBROUTINE statement is called a subroutine subprogram.

TABLE 4. BASIC EXTERNAL FUNCTIONS

Basic External Function	*Definition*	*Number of Arguments*	*Symbolic Name*	*Type of:*	
				Argument	*Function*
Exponential	e^a	1	EXP	Real	Real
		1	DEXP	Double	Double
		1	CEXP	Complex	Complex
Natural Logarithm	$\log_e(a)$	1	ALOG	Real	Real
		1	DLOG	Double	Double
		1	CLOG	Complex	Complex
Common Logarithm	$\log_{10}(a)$	1	ALOG10	Real	Real
			DLOG10	Double	Double
Trigonometric Sine	$\sin(a)$	1	SIN	Real	Real
		1	DSIN	Double	Double
		1	CSIN	Complex	Complex
Trigonometric Cosine	$\cos(a)$	1	COS	Real	Real
		1	DCOS	Double	Double
		1	CCOS	Complex	Complex
Hyperbolic Tangent	$\tanh(a)$	1	TANH	Real	Real
Square Root	$(a)^{1/2}$	1	SQRT	Real	Real
		1	DSQRT	Double	Double
		1	CSQRT	Complex	Complex
Arctangent	$\arctan(a)$	1	ATAN	Real	Real
		1	DATAN	Double	Double
	$\arctan(a_1/a_2)$	2	ATAN2	Real	Real
		2	DATAN2	Double	Double
Remaindering*	$a_1 \pmod{a_2}$	2	DMOD	Double	Double
Modulus		1	CABS	Complex	Real

* The function DMOD (a_1, a_2) is defined as $a_1 - [a_1/a_2]a_2$, where $[x]$ is the integer whose magnitude does not exceed the magnitude of x and whose sign is the same as the sign of x.

8.4.1. *Defining Subroutine Subprograms.* A SUBROUTINE statement is of one of the forms:

SUBROUTINE s $(a_1, a_2, \ldots, a_n)$
or
SUBROUTINE s

where:

1. s is the symbolic name of the subroutine to be defined.
2. The a's, called the dummy arguments, are each either a variable name, an array name, or an external procedure name.

Subroutine subprograms are constructed as specified in 9.1.3 with the following restrictions:

1. The symbolic name of the subroutine must not appear in any statement in this subprogram except as the symbolic name of the subroutine in the SUBROUTINE statement itself.
2. The symbolic names of the dummy arguments may not appear in an EQUIVALENCE, COMMON, or DATA statement in the subprogram.
3. The subroutine subprogram may define or redefine one or more of its arguments so as to effectively return results.
4. The subroutine subprogram may contain any statements except BLOCK DATA, FUNCTION, another SUBROUTINE statement, or any statement that directly or indirectly references the subroutine being defined.
5. The subroutine subprogram must contain at least one RETURN statement.

8.4.2. *Referencing Subroutines.* A subroutine is referenced by a CALL statement (7.1.2.4). The actual arguments, which constitute the argument list, must agree in order, number, and type with the corresponding dummy arguments in the defining program. The use of a Hollerith constant as an actual argument is an exception to the rule requiring agreement of type. An actual argument in a subroutine reference may be one of the following:

1. A Hollerith constant.
2. A variable name.
3. An array element name.
4. An array name.
5. Any other expression.
6. The name of an external procedure.

If an actual argument is an external function name or a subroutine name the corresponding dummy argument must be used as an external function name or a subroutine name, respectively.

If an actual argument corresponds to a dummy argument that is defined or redefined in the referenced subprogram the actual argument must be a variable name, an array element name, or an array name.

Execution of a subroutine reference as described in the foregoing results in an association of actual arguments with all appearances of dummy arguments in executable statements, function definition statements, and as adjustable dimensions in the defining subprogram. If the actual argument is as specified in item (5) in the foregoing this association is by value rather than by name. Following these associations, execution of the first executable statement of the defining subprogram is undertaken.

An actual argument which is an array element name containing variables in the subscript could in every case be replaced by the same argument with a constant subscript containing the same values as would be derived by computing the variable subscript just before the association of arguments takes place.

If a dummy argument of an external function is an array name the corresponding actual argument must be an array name or array element name (10.1.3).

If a subroutine reference causes a dummy argument in the referenced subroutine to become associated with another dummy argument in the same subroutine or with an entity in common, a definition of either entity within the subroutine is prohibited.

Unless it is a dummy argument, a subroutine is also referenced (in that it

must be defined) by the appearance of its symbolic name in an EXTERNAL statement.

8.5. BLOCK DATA SUBPROGRAM. A BLOCK DATA statement is of the form:

BLOCK DATA

This statement may only appear as the first statement of specification subprograms that are called block data subprograms, and that are used to enter initial values into elements of labelled common blocks. This special subprogram contains only type-statements, EQUIVALENCE, DATA, DIMENSION, and COMMON statements.

If any entity of a given common block is being given an initial value in such a subprogram a complete set of specification statements for the entire block must be included, even though some of the elements of the block do not appear in DATA statements. Initial values may be entered into more than one block in a single subprogram.

9. PROGRAMS

An executable program is a collection of statements, comment lines, and end lines that completely (except for input data values and their effects) describe a computing procedure.

9.1. PROGRAM COMPONENTS. Programs consist of program parts, program bodies, and subprogram statements.

9.1.1. *Program Part.* A program part must contain at least one executable statement and may contain FORMAT statements, and data initialization statements. It need not contain any statements from either of the latter two classes of statement. This collection of statements may optionally be preceded by statement function definitions, data initialization statements, and FORMAT statements. As before only some or none of these need be present.

9.1.2. *Program Body.* A program body is a collection of specification statements, FORMAT statements or both, or neither, followed by a program part, followed by an end line.

9.1.3. *Subprogram.* A subprogram consists of a SUBROUTINE or FUNCTION statement followed by a program body, or is a block data subprogram.

9.1.4. *Block Data Subprogram.* A block data subprogram consists of a BLOCK DATA statement, followed by the appropriate (8.5) specification statements, followed by data initialization statements, followed by an end line.

9.1.5. *Main Program.* A main program consists of a program body.

9.1.6. *Executable Program.* An executable program consists of a main program plus any number of subprograms, external procedures, or both.

9.1.7. *Program Unit.* A program unit is a main program or a subprogram.

9.2. NORMAL EXECUTION SEQUENCE. When an executable program begins operation execution commences with the execution of the first executable statement of the main program. A subprogram, when referenced, starts execution with execution of the first executable statement of that subprogram. Unless a statement is a GO TO, arithmetic IF, RETURN, or STOP statement or the terminal statement of a DO, completion of execution of that statement causes execution of the next following executable statement. The sequence of execution following execution of any of these statements is described in Section 7. A program part may not contain an executable statement that can never be executed.

A program part must contain a first executable statement.

10. INTRA- AND INTERPROGRAM RELATIONSHIPS

10.1. SYMBOLIC NAMES. A symbolic name has been defined to consist of from one to six alphanumeric characters, the first of which must be alphabetic. Sequences of characters that are format field descriptors or uniquely identify certain statement types, e.g. GO TO, READ, FORMAT, etc., are not symbolic names in such occurrences nor do they form the first characters of symbolic names in these cases. In a program unit a symbolic name (perhaps qualified by a subscript) must identify an element of one (and usually only one) of the following classes:

Class I An array and the elements of that array.
Class II A variable.
Class III A statement function.
Class IV An intrinsic function.
Class V An external function.
Class VI A subroutine.
Class VII An external procedure which cannot be classified as either a subroutine or an external function in the program unit in question.
Class VIII A block name.

10.1.1. *Restrictions on Class.* A symbolic name in Class VIII in a program unit may also be in any one of the Classes I, II, or III in that program unit.

In the program unit in which a symbolic name in Class V appears immediately following the word FUNCTION in a FUNCTION statement, that name must also be in Class II.

Once a symbolic name is used in Class V, VI, VII, or VIII in any unit of an executable program, no other program unit of that executable program may use that name to identify an entity of these classes other than the one originally identified. In the totality of the program units that make up an executable program a Class VII name must be associated with a Class V or VI name. Class VII can only exist locally in program units.

In a program unit no symbolic name can be in more than one class except as noted in the foregoing. There are no restrictions on uses of symbolic names in different program units of an executable program other than those noted in the foregoing.

10.1.2. *Implications of Mentions in Specification and* DATA *Statements.* A symbolic name is in Class I if, and only if, it appears as a declarator name. Only one such appearance for a symbolic name in a program unit is permitted.

A symbolic name that appears in a COMMON statement (other than as a block name) is either in Class I or in Class II but not Class V. (8.3.1) Only one such appearance for a symbolic name in a program unit is permitted.

A symbolic name that appears in an EQUIVALENCE statement is either in Class I or in Class II but not Class V. (8.3.1)

A symbolic name that appears in a type-statement cannot be in Class VI or Class VII. Only one such appearance for a symbolic name in a program unit is permitted.

A symbolic name that appears in an EXTERNAL statement is in either Class V, Class VI, or Class VII. Only one such appearance for a symbolic name in a program unit is permitted.

A symbolic name that appears in a DATA statement is in either Class I or in Class II but not Class V. (8.3.1) In an executable program, a storage unit (7.2.1.3.1) may have its value initialized one time at the most.

10.1.3. *Array and Array Element.* In a program unit any appearance of a symbolic name that identifies an array must be immediately followed by a subscript, except for the following cases:

1. In the list of an input/output statement.
2. In a list of dummy arguments.
3. In the list of actual arguments in a reference to an external procedure.
4. In a COMMON statement.
5. In a type-statement.

Only when an actual argument of an external procedure reference is an array name or an array element name may the corresponding dummy argument be an array name. If the actual argument is an array name the length of the dummy argument array must be no greater than the length of the actual argument array. If the actual argument is an array element name the length of the dummy argument array must be less than or equal to the length of the actual argument array plus one minus the value of the subscript of the array element.

10.1.4. *External Procedures.* The only case when a symbolic name is in Class VII occurs when that name appears only in an EXTERNAL statement and as an actual argument to an external procedure in a program unit.

Only when an actual argument of an external procedure reference is an external procedure name may the corresponding dummy argument be an external procedure name.

In the execution of an executable program a procedure subprogram may not be referenced twice without the execution of a RETURN statement in that procedure having intervened.

10.1.5. *Subroutine.* A symbolic name is in Class VI if it appears:

1. Immediately following the word SUBROUTINE in a SUBROUTINE statement.
2. Immediately following the word CALL in a CALL statement.

10.1.6. *Statement Function.* A symbolic name is in Class III in a program unit if, and only if, it meets all three of the following conditions:

1. It does not appear in an EXTERNAL statement nor is it in Class I.
2. Every appearance of the name, except in a type-statement, is immediately followed by a left parenthesis.
3. A function defining statement (8.1.1) is present for that symbolic name.

10.1.7. *Intrinsic Function.* A symbolic name is in Class IV in a program unit if, and only if, it meets all four of the following conditions:

1. It does not appear in an EXTERNAL statement nor is it in Class I or Class III.
2. The symbolic name appears in the name column of the table in Section 8.2.
3. The symbolic name does not appear in a type-statement of type different from the intrinsic type specified in the table.
4. Every appearance of the symbolic name (except in a type-statement as described in the foregoing) is immediately followed by an actual argument list enclosed in parentheses.

The use of an intrinsic function in a program unit of an executable program does not preclude the use of the same symbolic name to identify some other entity in a different program unit of that executable program.

10.1.8. *External Function.* A symbolic name is in Class V if it:

1. Appears immediately following the word FUNCTION in a FUNCTION statement.

2. Is not in Class I, Class III, Class IV, or Class VI and appears immediately followed by a left parenthesis on every occurrence except in a type-statement in an EXTERNAL statement, or as an actual argument. There must be at least one such appearance in the program unit in which it is so used.

10.1.9. *Variable.* In a program unit, a symbolic name is in Class II if it meets all three of the following conditions:

1. It is not in Class VI or Class VII.

2. It is never immediately followed by a left parenthesis unless it is immediately preceded by the word FUNCTION in a FUNCTION statement.

3. It occurs other than in a Class VIII appearance.

10.1.10. *Block Name.* A symbolic name is in Class VIII if, and only if, it is used as a block name in a COMMON statement.

10.2. DEFINITION. There are two levels of definition of numeric values, first-level definition and second-level definition. The concept of definition on the first level applies to array elements and variables; that of second-level definition to integer variables only. These concepts are defined in terms of progression of execution; and thus, an executable program, complete and in execution, is assumed in what follows.

There are two other varieties of definition that should be noted. The first, effected by GO TO assignment and referring to an integer variable being defined with other than an integer value, is discussed in 7.1.1.3 and 7.1.2.1.2; the second, which refers to when an external procedure may be referenced, will be discussed in the next section.

In what follows, otherwise unqualified use of the terms definition and undefinition (or their alternate forms) as applied to variables and array elements will imply modification by the phrase on the first level.

10.2.1. *Definition of Procedures.* If an executable program contains information describing an external procedure such an external procedure with the applicable symbolic name is defined for use in that executable program. An external function reference or subroutine reference (as the case may be) to that symbolic name may then appear in the executable program, provided that number of argument agrees between definition and reference. In addition, for an external function, the type of function must agree between definition and reference. Other restrictions on agreements are contained in 8.3.1, 8.3.2, 8.4.1, 8.4.2, 10.1.3, and 10.1.4.

The basic external functions listed in (8.3.3) are always defined and may be referenced subject to the restrictions alluded to in the foregoing.

A symbolic name in Class III or Class IV is defined for such use.

10.2.2. *Associations that Effect Definition.* Entities may become associated by:

1. COMMON association.
2. EQUIVALENCE association.
3. Argument substitution.

Multiple association to one or more entities can be the result of combinations of the foregoing. Any definition or undefinition of one of a set of associated entities effects the definition or undefinition of each entity of the entire set.

For purposes of definition, in a program unit there is no association between any two entities both of which appear in COMMON statements. Further, there

is no other association for common and equivalenced entities other than those stated in 7.2.1.3.1 and 7.2.1.4.

If an actual argument of an external procedure reference is an array name, an array element name, or a variable name, then the discussions in 10.1.3 and 10.2.1 allow an association of dummy arguments with the actual arguments only between the time of execution of the first executable statement of the procedure and the inception of execution of the next encountered RETURN statement of that procedure. Note specifically that this association can be carried through more than one level of external procedure reference.

In what follows, variables or array elements associated by the information in 7.2.1.3.1 and 7.2.1.4 will be equivalent if, and only if, they are of the same type.

If an entity of a given type becomes defined, then all associated entities of different type become undefined at the same time, while all associated entities of the same type become defined unless otherwise noted.

Association by argument substitution is valid only in the case of identity of type, so the rule in this case is that an entity created by argument substitution is defined at time of entry if, and only if, the actual argument was defined. If an entity created by argument substitution becomes defined or undefined (while the association exists) during execution of a subprogram, then the corresponding actual entities in all calling program units becomes defined or undefined accordingly.

10.2.3. *Events That Effect Definition.* Variables and array elements become initially defined if, and only if, their names are associated in a data initialization statement with a constant of the same type as the variable or array in question. Any entity not initially defined is undefined at the time of the first execution of the first executable statement of the main program. Redefinition of a defined entity is always permissible except for certain integer variables (7.1.2.8, 7.1.3.1.1, and 7.2.1.1.2) or certain entities in subprograms (6.4, 8.3.2, and 8.4.2).

Variables and array elements become defined or redefined as follows:

1. Completion of execution of an arithmetic or logical assignment statement causes definition of the entity that precedes the equals.
2. As execution of an input statement proceeds, each entity, which is assigned a value of its corresponding type from the input medium, is defined at the time of such association. Only at the completion of execution of the statement do associated entities of the same type become defined.
3. Completion of execution of a DO statement causes definition of the control variable.
4. Inception of execution of action specified by a DO-implied list causes definition of the control variable.

Variables and array elements become undefined as follows:

1. At the time a DO is satisfied, the control variable becomes undefined.
2. Completion of execution of an ASSIGN statement causes undefinition of the integer variable in the statement.
3. Certain entities in function subprograms (10.2.9) become undefined.
4. Completion of execution of action specified by a DO-implied list causes undefinition of the control variable.
5. When an associated entity of different types becomes defined.
6. When an associated entity of the same type becomes undefined.

10.2.4. *Entities in Blank Common.* Entities in blank common and those entities associated with them may not be initially defined.

Such entities, once defined by any of the rules previously mentioned, remain defined until they become undefined.

10.2.5. *Entities in Labelled Common.* Entities in labelled common or any associates of those entities may be initially defined.

A program unit contains a labelled common block name if the name appears as a block name in the program unit. If a main program or referenced subprogram contains a labelled common block name any entity in the block (and its associates) once defined remain defined until they become undefined.

It should be noted that redefinition of an initially defined entity will allow later undefinition of that entity. Specifically, if a subprogram contains a labelled common block name that is not contained in any program unit currently referencing the subprogram directly or indirectly, the execution of a RETURN statement in the subprogram causes undefinition of all entities in the block (and their associates) except for initially defined entities that have maintained their initial definitions.

10.2.6. *Entities Not in Common.* An entity not in common except for a dummy argument or the value of a function may be initially defined.

Such entities, once defined by any of the rules previously mentioned, remain defined until they become undefined.

If such an entity is in a subprogram the completion of execution of a RETURN statement in that subprogram causes all such entities and their associates at that time (except for initially defined entities that have not been redefined or become undefined) to become undefined. In this respect, it should be noted that the association between dummy arguments and actual arguments is terminated at the inception of execution of the RETURN statement.

Again, it should be emphasized, the redefinition of an initially defined entity can result in a subsequent undefinition of that entity.

10.2.7. *Basic Block.* In a program unit, a basic block is a group of one or more executable statements defined as follows.

The following statements are block terminal statements:

1. DO statement.
2. CALL statement.
3. GO TO statement of all types.
4. Arithmetic IF statement.
5. STOP statement.
6. RETURN statement.
7. The first executable statement, if it exists, preceding a statement whose label is mentioned in a GO TO or arithmetic IF statement.
8. An arithmetic statement in which an integer variable precedes the equals.
9. A READ statement with an integer variable in the list.
10. A logical IF containing any of the admissible forms given in the foregoing.

The following statements are block initial statements:

1. The first executable statement of a program unit.
2. The first executable statement, if it exists, following a block terminal statement.

Every block initial statement defines a basic block. If that initial statement is also a block terminal statement the basic block consists of that one statement. Otherwise, the basic block consists of the initial statement and all executable statements that follow until a block terminal statement is encountered. The terminal statement is included in the basic block.

10.2.7.1. *Last Executable Statement.* In a program unit the last executable statement (which cannot be part of a logical IF) must be one of the following statements: GO TO statement, arithmetic IF statement, STOP statement, or RETURN statement.

10.2.8. *Second Level Definition.* Integer variables must be defined on the second level when used in subscripts and computed GO TO statements.

Redefinition of an integer entity causes all associated variables to be undefined for use on the second level during this execution of this program unit until the associated integer variable is explicitly redefined.

Except as just noted, an integer variable is defined on the second level upon execution of the initial statement of a basic block only if both of the following conditions apply:

1. The variable is used in a subscript or in a computed GO TO in the basic block in question.
2. The variable is defined on the first level at the time of execution of the initial statement in question.

This definition persists until one of the following happens:

1. Completion of execution of the terminal statement of the basic block in question.
2. The variable in question becomes undefined or receives a new definition on the first level.

At this time the variable becomes undefined on the second level.

In addition, the occurrence of an integer variable in the list of an input statement in which that integer variable appears following in a subscript causes that variable to be defined on the second level. This definition persists until one of the following happens:

1. Completion of execution of the terminal statement of the basic block containing the input statement.
2. The variable becomes undefined or receives a new definition on the first level.

An integer variable defined as the control variable of a DO-implied list is defined on the second level over the range of that DO-implied list and only over that range.

10.2.9. *Certain Entities in Function Subprograms.* If a function subprogram is referenced more than once with an identical argument list in a single statement, the execution of that subprogram must yield identical results for those cases mentioned, no matter what the order of evaluation of the statement.

If a statement contains a factor that may not be evaluated (6.4), and if this factor contains a function reference, then all entities that might be defined in that reference become undefined at the completion of evaluation of the expression containing the factor.

10.3. Definition Requirements for Use of Entities. Any variable referenced in a subscript or a computed GO TO must be defined on the second level at the time of this use.

Any variable, array element, or function referenced as a primary in an expression and any subroutine referenced by a CALL statement must be defined at the time of this use. In the case where an actual argument in the argument list of an external procedure reference is a variable name or an array element name, this in itself is not a requirement that the entity be defined at the time of

the procedure reference; however, when such an argument is an external procedure name, it must be defined.

Any variable used as an initial value, terminal value, or incrementation value of a DO statement or a DO-implied list must be defined at the time of this use.

Any variable used to identify an input/output unit must be defined at the time of this use.

At the time of execution of a RETURN statement in a function subprogram, the value (8.3.1) of that function must be defined.

At the time of execution of an output statement, every entity whose value is to be transferred to the output medium must be defined unless the output is under control of a format specification and the corresponding conversion code is A. If the output is under control of a format specification a correct association of conversion code with type of entity is required unless the conversion code is A. The following are the correct associations: I with integer; D with double precision; E, F, and G with real and complex; and L with logical.

BASIC FORTRAN*

CONTENTS

1. Introduction

2. Basic Terminology

3. Program Form
3.1. The FORTRAN character set
3.2. Lines
3.3. Statements
3.4. Statement label
3.5. Symbolic names
3.6. Ordering of characters

4. Data Types
4.1. Data types association
4.2. Data type properties

5. Data and Procedure Identification
5.1. Data and procedure names
5.1.1. Constants
5.1.2. Variables
5.1.3. Array
5.1.4. Procedures
5.2. Function reference
5.3. Type rules for data and procedure identifiers
5.4. Dummy arguments

6. Expressions
6.1. Arithmetic expressions
6.4. Evaluation of expressions

7. Statements
7.1. Executable statements
7.1.1. Assignment statements
7.1.2. Control statements
7.1.2.1. GO TO statements
7.1.2.2. Arithmetic IF statement
7.1.2.4. CALL statement
7.1.2.5. RETURN statement
7.1.2.6. CONTINUE statement
7.1.2.7. Program control statements
7.1.2.8. DO statement
7.1.3. Input/Output statements
7.1.3.1. READ and WRITE statements
7.1.3.2. Auxiliary Input/Output statements
7.1.3.3. Printing of formatted records
7.2. Nonexecutable statements
7.2.1. Specification statements
7.2.1.1. Array declarator
7.2.1.2. DIMENSION statement
7.2.1.3. COMMON statement
7.2.1.4. EQUIVALENCE statement
7.2.3. FORMAT statement

8. Procedures and Suprograms
8.1. Statement functions
8.2. Intrinsic function and their reference
8.3. External functions
8.4. Subroutine

9. Programs
9.1. Program components
9.2. Normal execution sequence

10. Intra- and Interprogram Relationships
10.1. Symbolic names
10.2. Definition
10.3. Definition requirements for use of entities

* Popularly known as FORTRAN II (H.B.)

1. INTRODUCTION

1.1. PURPOSE. This specification establishes the form for and the interpretation of programs expressed in the FORTRAN language for the purpose of promoting a high degree of interchangeability of such programs for use on a variety of automatic data processing systems. A processor shall conform to this specification provided it accepts, and interprets as specified, at least those forms and relationships described herein.

Insofar as the interpretation of the form and relationships described are not affected, any statement of requirement could be replaced by a statement expressing that the specification does not provide an interpretation unless the requirement is met. Further, any statement of prohibition could be replaced by a statement expressing that the specification does not provide an interpretation when the prohibition is violated.

1.2. SCOPE. This specification establishes:

1. The form of a program written in the FORTRAN language.
2. The form of writing input data to be processed by such a program operating on automatic data processing systems.
3. Rules for interpreting the meaning of such a program.
4. The form of the output data resulting from the use of such a program on automatic data processing systems, provided that the rules of interpretation establish an interpretation.

This specification does not prescribe:

1. The mechanism by which programs are transformed for use on a data processing system (the combination of this mechanism and data processing system is called a processor).
2. The method of transcription of such programs or their input or output data to or from a data processing medium.
3. The manual operations required for set-up and control of the use of such programs on data processing equipment.
4. The results when the rules for interpretation fail to establish an interpretation of such a program.
5. The size or complexity of a program that will exceed the capacity of any specific data processing system or the capability of a particular processor.
6. The range of precision of numerical quantities.

2. BASIC TERMINOLOGY

This section introduces some basic terminology and some concepts. A rigorous treatment of these is given in later sections. Certain conventions concerning the meaning of grammatical forms and particular words are presented.

A program that can be used as a self-contained computing procedure is called an *executable program* (9.1.6).

An executable program consists of precisely one main program and possibly one or more subprograms (9.1.6).

A *main program* is a set of statements and comments not containing a FUNCTION or SUBROUTINE statement (9.1.5).

A *procedure subprogram* is similar to a main program but is headed by a FUNCTION or SUBROUTINE statement. A procedure subprogram is sometimes referred to as a subprogram (9.1.3).

The term *program unit* will refer to either a main program or subprogram (9.1.7).

Any program unit may reference an *external procedure* (Section 9).

An external procedure that is defined by FORTRAN statements is called a *procedure subprogram.* External procedures also may be defined by other means. An external procedure may be an external function or an external subroutine. An external function defined by FORTRAN statements headed by a FUNCTION statement is called a *function subprogram.* An external subroutine defined by FORTRAN statements headed by a SUBROUTINE statement is called a *subroutine subprogram* (Sections 8 and 9).

Any program unit consists of *statements* and *comments.* A statement is divided into physical sections called *lines*, the first of which is called an *initial line* and the rest of which are called *continuation lines* (3.2).

There is a type of line called a comment that is not a statement and merely provides information for documentary purposes (3.2).

The statements in FORTRAN fall into two broad classes—executable and nonexecutable. The executable statements specify the action of the program while the nonexecutable statements describe the use of the program, the characteristics of the operands, editing information, statement functions, or data arrangement (7.1, 7.2).

The syntactic elements of a statement are *names* and *operators.* Names are used to reference objects such as data or procedures. Operators, including the imperative verbs, specify action upon named objects.

One class of name, the *array name*, deserves special mention. The name and the dimensions of the array of values denoted by the array name are declared prior to use. An array name may be used to identify an entire array. An array name qualified by a subscript may be used to identify a particular element of the array (5.1.3).

Data names and the arithmetic operators may be connected into arithmetic expressions that develop values. These values are derived by performing the specified operations on the named data (Section 6).

The identifiers used in FORTRAN are names and numbers. Data are named. Procedures are named. Statements are labelled with numbers. Input/output units are numbered or identified by a name whose value is the numerical unit designation.

At various places in this document there are statements with associated lists of entries. In all such cases the list is assumed to contain at least one entry unless an explicit exception is stated. As an example, in the statement

$$\text{SUBROUTINE } s(a_1, a_2, \ldots, a_n)$$

it is assumed that at least one symbolic name is included in the list within parentheses. A *list* is a set of identifiable elements, each of which is separated from its successor by a comma. Further, in a sentence a plural form of a noun will be assumed to also specify the singular form of that noun as a special case when the context of the sentence does not prohibit this interpretation.

The term *reference* is used as a verb with special meaning as defined in Section 5.

3. PROGRAM FORM

Every program unit is constructed of characters grouped into lines and statements.

3.1. THE FORTRAN CHARACTER SET. A program unit is written using the following characters: 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, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, and:

Character	*Name of Character*
	Blank
=	Equals
+	Plus
−	Minus
*	Asterisk
/	Slash
(	Left Parenthesis
)	Right Parenthesis
,	Comma
.	Decimal Point

The order in which the characters are listed does not imply a collating sequence.

3.1.1. *Digits.* A digit is one of the ten characters: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Unless specified otherwise, a string of digits will be interpreted in the decimal base number system when a number system base interpretation is appropriate.

An octal digit is one of the eight characters: 0, 1, 2, 3, 4, 5, 6, 7. These are only used in the STOP (7.1.2.7.1) and PAUSE (7.1.2.7.2) statements.

3.1.2. *Letters.* A letter is one of the twenty-six characters: 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.

3.1.3. *Alphanumeric Characters.* An alphanumeric character is a letter or a digit.

3.1.4. *Special Characters.* A special character is one of the ten characters: blank, equals, plus, minus, asterisk, slash, left parenthesis, right parenthesis, comma, and decimal point.

3.1.4.1. *Blank Character.* With the exception of the uses specified (3.2.2, 3.2.3, 3.2.4, 7.2.3.6, and 7.2.3.8), a blank character has no meaning and may be used freely to improve the appearance of the program subject to the restriction on continuation lines in 3.3.

3.2. LINES. A line is a string of 72 characters. All characters must be from the FORTRAN character set except as described in 7.2.3.8.

The character positions in a line are called columns and are consecutively numbered 1, 2, 3, . . . , 72. The number indicates the sequential position of a character in the line starting at the left and proceeding to the right.

3.2.1. *Comment Line.* The line C in column 1 of a line designates that line as a comment line. A comment line must be immediately followed by an initial line, another comment line, or an end line.

A comment line does not affect the program in any way and is available as a convenience for the programmer.

3.2.2. *End Line.* An end line is a line with the character blank in columns 1 through 6, the characters E, N, and D, once each in that order, in columns 7 through 72, preceded by, interspersed with, or followed by the character blank. The end line indicates, to the processor, the end of the written description of a program unit (9.1.7). Every program unit must physically terminate with an end line.

3.2.3. *Initial Line.* An initial line is a line that is neither a comment line nor an end line and that contains the digit 0 or the character blank in column 6. Columns 1 through 5 contain the statement label or each contains the character blank.

3.2.4. *Continuation Line.* A continuation line is a line that contains any character other than the digit 0 or the character blank in column 6, and does not contain the character C in column 1.

A continuation line may only follow an initial line or another continuation line.

3.3. STATEMENTS. A statement consists of an initial line optionally followed by up to five ordered continuation lines. The statement is written in columns 7 through 72 of the lines. The order of the characters in the statement is columns 7 through 72 of the initial line followed, as applicable, by columns 7 through 72 of the first continuation line, columns 7 through 72 of the next continuation line, etc.

3.4. STATEMENT LABEL. Optionally, a statement may be labelled so that it may be referred to in other statements. A statement label consists of from one to four digits. The value of the integer represented is not significant but must be greater than zero. The statement label may be placed anywhere in columns 1 through 5 of the initial line of the statement. The same statement label may not be given to more than one statement in a program unit. Leading zeros are not significant in differentiating statement labels.

3.5. SYMBOLIC NAMES. A symbolic name consists of from one to five alphanumeric characters, the first of which must be alphabetic. See 10.1 through 10.1.10 for a discussion of classification of symbolic names and restrictions on their use.

3.6. ORDERING OF CHARACTERS. An ordering of characters is assumed within a program unit. Thus, any meaningful collection of characters that constitutes names, lines, and statements exists as a totally ordered set. This ordering is imposed by the character position rule of 3.2 (which orders characters within a line) and the order in which lines are presented for processing.

4. DATA TYPES

Two different types of data are defined. These are integer and real. Each type has a different mathematical significance and may have different internal representation. Thus, the data type has a significance in the interpretation of the associated operations with which a datum is involved. The data type of a function defines the type of the datum it supplies to the expression in which it appears.

4.1. DATA TYPE ASSOCIATION. The name employed to identify a datum or function carries the data type association. The form of the string representing a constant defines both the value and the data type.

A symbolic name representing a function, variable, or array must have only a single data type association for each program unit. Once associated with a particular data type, a specific name implies that type for any differing usage of that symbolic name that requires a data type association throughout the program unit in which it is defined.

Data type is established for a symbolic name by the first character of that name (5.3).

4.2. DATA TYPE PROPERTIES. The mathematical and the representation properties for each of the data types are defined in the following sections. For both real and integer data, the value zero is considered neither positive nor negative.

4.2.1. *Integer Type.* An integer datum is always an exact representation of an

integer value. It may assume positive, negative, and zero values. It may only assume integral values.

4.2.2. *Real Type.* A real datum is a processor approximation to the value of a real number. It may assume positive, negative, and zero values.

5. DATA AND PROCEDURE IDENTIFICATION

Names are employed to reference or otherwise identify data and procedures.

The term *reference* is used to indicate an identification of a datum implying that the current value of the datum will be made available during the execution of the statement containing the reference. If the datum is identified but not necessarily made available the datum is said to be *named.* One case of special interest in which the datum is named is that of assigning a value to a datum, thus defining or redefining the datum.

The term, reference, is used to indicate an identification of a procedure implying that the actions specified by the procedure will be made available.

A complete and rigorous discussion of reference and definition, including redefinition, is contained in Section 10.

5.1. DATA AND PROCEDURE NAMES. A data name identifies a constant, a variable, an array, or an array element. A procedure name identifies a function or a subroutine.

5.1.1. *Constants.* A constant is a name that references a value. A constant may not be redefined.

An integer or real constant is said to be signed when it is immediately preceded by a plus or minus. Also, for these types, an optionally signed constant is either a constant or a signed constant.

5.1.1.1. *Integer Constant.* An integer constant is formed by a nonempty string of digits. The datum formed this way is interpreted as the value represented by the digit string.

5.1.1.2. *Real Constant.* A basic real constant is formed by an integer part, a decimal point, and a decimal fraction part in that order. Both the integer part and the decimal fraction part are formed by a string of digits; either one of these strings may be empty, but not both. The datum formed this way is interpreted as representing a value that is an approximation to the number represented by the integer and fraction parts.

A decimal exponent is formed by the letter E followed by an optionally signed integer constant. This exponent is interpreted as a multiplier (to be applied to the constant immediately preceding it) that is an approximation to ten raised to the power specified by the field following the E.

A real constant is either a basic real constant or a basic real constant followed by a decimal exponent.

5.1.2. *Variable.* A variable is a datum that is identified by a symbolic name (3.5). Such a datum may be referenced and defined.

5.1.3. *Array.* An array is an ordered set of data of one or two dimensions. An array is identified by a symbolic name. Identification of the entire ordered set is achieved via use of the array name.

5.1.3.1. *Array Element.* An array element is one of the members of the set of data of an array. An array element is identified by immediately following the array name with a qualifier, called a subscript, which points to the particular element of the array.

An array element may be referenced and defined.

5.1.3.2. *Subscript.* A subscript is formed by a parenthesized list of subscript

expressions. The subscript expressions are separated by a comma if two are present. The number of subscript expressions must correspond to the declared dimensionality (7.2.1.1), except in an EQUIVALENCE statement (7.2.1.4). Following evaluation of all of the subscript expressions, the array element successor function (7.2.1.1.1) determines the identified array element.

5.1.3.3. *Subscript Expressions.* A subscript expression is formed from one of the following constructs:

$$c*v+k$$
$$c*v-k$$
$$c*v$$
$$v+k$$
$$v-k$$
$$v$$
$$k$$

where c and k are integer constants and v is an integer variable reference. See Section 6 for a discussion of evaluation of expressions and 10.2.8 and 10.3 for requirements that apply to the use of a variable in a subscript.

5.1.4. *Procedures.* A procedure (Section 8) is identified by a symbolic name. A procedure is a statement function, an intrinsic function, a basic external function, an external function, or an external subroutine. Statement functions, intrinsic functions, basic external functions, and external functions are referred to as functions or function procedures; external subroutines as subroutines or subroutine procedures.

A function supplies a result to be used at the point of reference; a subroutine does not. Functions are referenced in a manner different from subroutines.

5.2. FUNCTION REFERENCE. A function reference consists of the function name followed by an actual argument list enclosed in parentheses. If the list contains more than one argument, the arguments are separated by commas. The allowable forms of function arguments are given in Section 8.

See 10.2.1 for a discussion of requirements that apply to function references.

5.3. TYPE RULES FOR DATA AND PROCEDURE IDENTIFIERS. The type of a constant is implicit in its name.

There is no type associated with a symbolic name that identifies a subroutine.

A symbolic name that identifies a variable, an array, or a statement function has a type implied by the first character of the name: I, J, K, L, M, and N imply type integer; any other letter implies type real.

A symbolic name that identifies an intrinsic function or a basic external function when it is used to identify this designated procedure, has a type associated with it as specified in Tables 3 and 4.

In the program unit in which an external function is referenced or defined its type definition is defined in the same manner as for a variable and an array.

The same type is associated with an array element as is associated with the array name.

5.4. DUMMY ARGUMENTS. A dummy argument of an external procedure identifies a variable or an array.

Unless specified otherwise, when the use of a variable, array, or array element name is specified, the use of a dummy argument is permissible provided that a proper association with an actual argument is made.

The process of argument association is discussed in Sections 8 and 10.

6. EXPRESSIONS

This section gives the formation and evaluation rules for arithmetic expressions. An expression is formed from elements and operators. See 10.3 for a discussion of requirements that apply to the use of certain entities in expressions.

6.1. ARITHMETIC EXPRESSIONS. An arithmetic expression is formed with arithmetic operators and arithmetic elements. Both the expression and its constituent elements identify values of one of the types integer or real. The arithmetic operators are:

Operator	*Representing*
+	Addition, positive value (zero + element)
−	Subtraction, negative value (zero − element)
*	Multiplication
/	Division
**	Exponentiation

The arithmetic elements are primary, factor, term, signed term, simple arithmetic expression, and arithmetic expression.

A primary is an arithmetic expression enclosed in parentheses, a constant, a variable reference, an array element reference, or a function reference.

A factor is a primary or a construct of the form:

*primary**primary*

A term is a factor or a construct of one of the forms:

term/factor
or
*term*term*

A signed term is a term immediately preceded by + or −.

A simple arithmetic expression is a term or two simple arithmetic expressions separated by a + or −.

An arithmetic expression is a simple arithmetic expression or a signed term or either of the preceding forms immediately followed by a + or − immediately followed by a simple arithmetic expression.

A primary of any type may be exponentiated by an integer primary and the resultant factor is of the same type as that of the element being exponentiated. A real primary may be exponentiated by a real primary, and the resultant factor is of type real. These are the only cases for which use of the exponentiation operator is defined.

By use of the arithmetic operators other than exponentiation, any admissible element may be combined with another admissible element of the same type, and the resultant element is of the same type.

6.4. EVALUATION OF EXPRESSIONS. A part of an expression need be evaluated only if such action is necessary to establish the value of the expression. The rules for formation of expressions imply the binding strength of operators. It should be noted that the range of the subtraction operator is the term that immediately succeeds it. The evaluation may proceed according to any valid formation sequence.

When two elements are combined by an operator the order of evaluation of the elements is optional. If mathematical use of operators is associative, commutative, or both, full use of these facts may be made to revise orders of combination, provided only that integrity of parenthesized expressions is not violated.

The value of an integer factor or term is the nearest integer whose magnitude does not exceed the magnitude of the mathematical value represented by that factor or term.

Any use of an array element name requires the evaluation of its subscript. The evaluation of functions appearing in an expression may not validly alter the value of any other element within the expressions, assignment statement or CALL statement in which the function reference appears. The type of the expression in which a function reference or subscript appears does not affect, nor is it affected by, the evaluation of the actual arguments or subscript.

No factor may be evaluated that requires a negative valued primary to be raised to a real exponent. No factor may be evaluated that requires raising a zero valued primary to a zero valued exponent.

No element may be evaluated whose value is not mathematically defined.

7. STATEMENTS

A statement may be classified as executable or nonexecutable. Executable statements specify actions; nonexecutable statements describe the characteristics and arrangement of data, editing information, statement functions, and classification of program units.

7.1. EXECUTABLE STATEMENTS. There are three types of executable statements:

1. Assignment statements.
2. Control statements.
3. Input/output statements.

7.1.1. *Assignment Statements.* There is a single assignment statement, the arithmetic assignment statement.

7.1.1.1. *Arithmetic Assignment Statement.* An arithmetic assignment statement is of the form:

$$v = e$$

where v is a variable name or array element name and e is an arithmetic expression. Execution of this statement causes the evaluation of the expression e and the altering of v according to Table 1.

TABLE 1. RULES FOR ASSIGNMENT OF e TO v

If v Type Is	*And e Type Is*	*The Assignment Rule Is**
Integer	Integer	Assign
Integer	Real	Fix & Assign
Real	Integer	Float & Assign
Real	Real	Assign

* NOTES.

1. Assign means transmit the resulting value, without change, to the entity.
2. Fix means truncate any fractional part of the result and transform that value to the form of an integer datum.
3. Float means transform the value to the form of a real datum.

7.1.2. *Control Statements.* There are seven types of control statements:

1. GO TO statements.
2. Arithmetic IF statement.
3. CALL statement.
4. RETURN statement.
5. CONTINUE statement.
6. Program control statements.
7. DO statement.

The statement labels used in a control statement must be associated with executable statements within the same program unit in which the control statement appears.

7.1.2.1. GO TO *Statements.* There are two types of GO TO statements:

1. Unconditional GO TO statement.
2. Computed GO TO statement.

7.1.2.1.1. *Unconditional* GO TO *Statement.* An unconditional GO TO statement is of the form:

GO TO k

where k is a statement label.

Execution of this statement causes the statement identified by the statement label to be executed next.

7.1.2.1.3. *Computed* GO TO *Statement.* A computed GO TO statement is of the form:

GO TO $(k_1, k_2, \ldots, k_n), i$

where the k's are statement labels and i is an integer variable reference. See 10.2.8 and 10.3 for a discussion of requirements that apply to the use of a variable in a computed GO TO statement.

Execution of this statement causes the statement identified by the statement label k_j to be executed next, where j is the value of i at the time of the execution. This statement is defined only for values such that $1 \leqslant j \leqslant n$.

7.1.2.2. *Arithmetic* IF *Statement.* An arithmetic IF statement is of the form:

IF (e) k_1, k_2, k_3

where e is any arithmetic expression of type integer or real, and the k's are statement labels.

The arithmetic IF is a three-way branch. Execution of this statement causes evaluation of the expression e following which the statement identified by the statement label k_1, k_2, or k_3 is executed next as the value of e is less than zero, zero, or greater than zero, respectively.

7.1.2.4. CALL *Statement.* A CALL statement is of one of the forms:

CALL s $(a_1, a_2, \ldots, a_n)$
or
CALL s

where s is the name of a subroutine and the a's are actual arguments (8.4.2).

The inception of execution of a CALL statement references the designated subroutine. Return of control from the designated subroutine completes execution of the CALL statement.

7.1.2.5. RETURN *Statement.* A RETURN statement is of the form:

RETURN

A RETURN statement marks the logical end of a procedure subprogram and, thus, may only appear in a procedure subprogram.

Execution of this statement when it appears in a subroutine subprogram causes return of control to the current calling program unit.

Execution of this statement when it appears in a function subprogram causes return of control to the current calling program unit. At this time the value of the function (8.3.1) is made available.

7.1.2.6. CONTINUE *Statement.* A CONTINUE statement is of the form:

CONTINUE

Execution of this statement causes continuation of normal execution sequence.

7.1.2.7. *Program Control Statements.* There are two types of program control statements:

1. STOP statement
2. PAUSE statement

7.1.2.7.1. STOP *Statement.* A STOP statement is of one of the forms:

STOP n
or
STOP

where n is an octal digit string of length from one to four.

Execution of this statement causes termination of execution of the executable program.

7.1.2.7.2. PAUSE *Statement.* A PAUSE statement is of one of the forms:

PAUSE n
or
PAUSE

where n is an octal digit string of length from one to four.

The inception of execution of this statement causes a cessation of execution of this executable program. Execution must be resumable. At the time of cessation of execution, the octal digit string is accessible. The decision to resume execution is not under control of the program; but if execution is resumed, execution of the PAUSE statement is completed.

7.1.2.8. DO *Statement.* A DO statement is of one of the forms:

DO n $i = m_1, m_2, m_3$
or
DO n $i = m_1, m_2$

where:

1. n is the statement label of an executable statement. This statement, called the terminal statement of the associated DO, must physically follow and be in the same program unit as that DO statement. The terminal statement may not be a GO TO of any form, arithmetic IF, RETURN, STOP, PAUSE, or DO statement.
2. i is an integer variable name; this variable is called the control variable.
3. m_1, called the initial parameter ;m_2, called the terminal parameter; and m_3, called the incrementation parameter, are each either an integer constant or

integer variable reference. If the second form of the DO statement is used so that m_3 is not explicitly stated, a value of one is implied for the incrementation parameter. At time of execution of the DO statement, m_1, m_2, and m_3 must be greater than zero.

Associated with each DO statement is a range that is defined to be those executable statements from and including the first executable statement following the DO, to and including the terminal statement associated with the DO. A special situation occurs when the range of a DO contains another DO statement. In this case, the range of the contained DO must be a subset of the range of the containing DO.

A DO statement is used to define a loop. The action succeeding execution of a DO statement is described by the following five steps:

1. The control variable is assigned the value represented by the initial parameter. This value must be less than or equal to the value represented by the terminal parameter.
2. The range of the DO is executed.
3. If control reaches the terminal statement, and after execution of the terminal statement, the control variable of the most recently executed DO statement associated with the terminal statement is incremented by the value represented by the associated incrementation parameter.
4. If the value of the control variable after incrementation is less than or equal to the value represented by the associated terminal parameter the action as described starting at step 2 is repeated with the understanding that the range in question is that of the DO, the control variable of which was most recently incremented. If the value of the control variable is greater than the value represented by its associated terminal parameter the DO is said to have been satisfied and the control variable becomes undefined.
5. At this point, if there were one or more other DO statements referring to the terminal statement in question the control variable of the next most recently executed DO statement is incremented by the value represented by its associated incrementation parameter and the action as described in step 4 is repeated until all DO statements referring to the particular termination statement are satisfied, at which time the first executable statement following the terminal statement is executed.

Upon exiting from the range of a DO by execution of a GO TO statement or an arithmetic IF statement, that is, other than by satisfying the DO, the control variable of the DO is defined and is equal to the most recent value attained as defined in the foregoing.

A GO TO statement or an arithmetic IF statement may not cause control to pass into the range of a DO from outside its range. When a procedure reference occurs in the range of a DO, the actions of that procedure are considered to be temporarily within that range, i.e. during the execution of that reference.

The control variable, initial parameter, terminal parameter, and incrementation parameters of a DO may not be redefined during the execution of the range of that DO.

If a statement is the terminal statement of more than one DO statement the statement label of that terminal statement may not be used in any GO TO or arithmetic IF statement that occurs anywhere but in the range of the most deeply contained DO with that terminal statement.

7.1.3. *Input/Output Statements.* There are two types of input/output statements:

1. READ and WRITE statements.
2. Auxiliary input/output statements.

The first type consists of the statements that cause transfer of records of sequential files to and from internal storage, respectively. The second type consists of the BACKSPACE and REWIND statements that provide for positioning of such an external file, and ENDFILE, which provides for demarcation of such an external file.

In the following descriptions *u* and *f* identify input/output units and format specifications, respectively. An input/output unit is identified by an integer value and *u* may be either an integer constant or an integer variable reference whose value then identifies the unit. The format specification is described in 7.2.3. The statement label of a FORMAT statement is represented by *f*. The identified statement must appear in the same program unit as the input/output statement.

A particular unit has a single sequential file associated with it. The most general case of such a unit has the following properties:

1. If the unit contains one or more records, those records exist as a totally ordered set.
2. There exists a unique position of the unit called its initial point. If a unit contains no records, that unit is positioned at its initial point. If the unit is at its initial point and contains records, the first record of the unit is defined as the next record.
3. If a unit is not positioned at its initial point, there exists a unique preceding record associated with that position. The least of any records in the ordering described by (1) following this preceding record is defined as the next record of that position.
4. Upon completion of execution of a WRITE or ENDFILE statement, there exist no records following the records created by that statement.
5. When the next record is transmitted, the position of the unit is changed so that this next record becomes the preceding record.

If a unit does not provide for some of the properties given in the preceding, certain statements that will be defined may not refer to that unit. The use of such a statement is not defined for that unit.

7.1.3.1. READ *and* WRITE *Statements.* The READ and WRITE statements specify transfer of information. Each such statement may include a list of the names of variables, arrays, and array elements. The named elements are assigned values on input and have their values transferred on output.

Records may be formatted or unformatted. A formatted record consists of a string of characters. The transfer of such a record requires that a format specification be referenced to supply the necessary positioning and conversion specifications (7.2.3). The number of records transferred by the execution of a formatted READ or WRITE is dependent upon the list and referenced format specification (7.2.3.4). An unformatted record consists of a string of values. When an unformatted or formatted READ statement is executed, the required records on the identified unit must be, respectively, unformatted or formatted records.

7.1.3.1.1. *Input/Output Lists.* The input list specifies the names of the variables and array elements to which values are assigned on input. The output list specifies the references to variables and array elements whose values are transmitted. The input and output lists are of the same form.

Lists are formed in the following manner. A simple list is a variable name, an array element name, or an array name, or two simple lists separated by a comma.

A list is a simple list, a simple list enclosed in parentheses, a DO-implied list, or two lists separated by commas.

A DO-implied list is a list followed by a comma and a DO-implied specification, all enclosed in parentheses.

A DO-implied specification is of one of the forms:

$$i = m_1, m_2, m_3$$
or
$$i = m_1, m_2$$

The elements i, m_1, m_2, and m_3 are as defined for the DO statement (7.1.2.8). The range of DO-implied specification is the list of the DO-implied list and, for input lists, i, m_1, m_2, and m_3 may appear, within that range, only in subscripts.

A variable name or array element name specifies itself. An array name specifies all of the array element names defined by the array declarator, and they are specified in the order given by the array element successor function (7.2.1.1.1).

The elements of a list are specified in the order of their occurrence from left to right. The elements of a list in a DO-implied list are specified for each cycle of the implied DO.

7.1.3.1.2. *Formatted* READ. A formatted READ statement is of one of the forms:

READ $(u, f)k$
or
READ (u, f)

where k is a list.

Execution of this statement causes the input of the next records from the unit identified by u. The information is scanned and converted as specified by the format specification identified by f and the resulting values are assigned to the elements specified by the list. See, however, 7.2.3.4.

7.1.3.1.3. *Formatted* WRITE. A formatted WRITE statement is of one of the forms:

WRITE $(u, f)k$
or
WRITE (u, f)

where k is a list.

Execution of this statement creates the next records on the unit identified by u. The list specifies a sequence of values, and these are converted and positioned as specified by the format specification identified by f. See, however, 7.2.3.4.

7.1.3.1.4. *Unformatted* READ. An unformatted READ statement is of one of the forms:

READ $(u)k$
or
READ (u)

where k is a list.

Execution of this statement causes the input of the next record from the unit identified by u, and, if there is a list, these values are assigned to the sequence of elements specified by the list. The sequence of values required by the list may not exceed the sequence of values from the unformatted record.

7.1.3.1.5. *Unformatted* WRITE. An unformatted WRITE statement is of the form:

WRITE $(u)k$

where k is a list.

Execution of this statement creates the next record on the unit identified by u of the sequence of values specified by the list.

7.1.3.2. *Auxiliary Input/Output Statements.* There are three types of auxiliary input/output statements:

1. REWIND statement.
2. BACKSPACE statement.
3. ENDFILE statement.

7.1.3.2.1. REWIND *Statement.* A REWIND statement is of the form:

REWIND u

Execution of this statement causes the unit identified by u to be positioned at its initial point.

7.1.3.2.2. BACKSPACE *Statement.* A BACKSPACE statement is of the form:

BACKSPACE u

If the unit identified by u is positioned at its initial point, execution of this statement has no effect. Otherwise, the execution of this statement results in the positioning of the unit identified by u so that what had been the preceding record prior to that execution becomes the next record.

7.1.3.2.3. ENDFILE *Statement.* An ENDFILE statement is of the form:

ENDFILE u

Execution of this statement causes the recording of an endfile record on the unit identified by u. The endfile record is a unique record signifying a demarcation of a sequential file. Action is undefined when an endfile record is encountered during execution of a READ statement.

7.1.3.3. *Printing of Formatted Records.* When formatted records are prepared for printing the first character of such a record is not printed.

7.2. NONEXECUTABLE STATEMENTS. There are four types of nonexecutable statements:

1. Specification statements.
2. FORMAT statement.
3. Function defining statements.
4. Subprogram statements.

See 10.1.2 for a discussion of restrictions on appearances of symbolic names in such statements.

The function defining statements and subprogram statements are discussed in Section 8.

7.2.1. *Specification Statements.* There are three types of specification statements:

1. DIMENSION statement.
2. COMMON statement.
3. EQUIVALENCE statement.

7.2.1.1. *Array Declarator.* An array declarator specifies an array used in a program unit.

The array declarator indicates the symbolic name, the number of dimensions (one or two), and the size of each of the dimensions. The array declarator statement is the DIMENSION statement.

An array declarator has the form:

$$v\ (i)$$

where:

1. v, called the declarator name, is a symbolic name.
2. (i), called the declarator subscript, is composed of an integer constant or two integer constants separated by a comma.

The appearance of a declarator in a declarator statement serves to inform the processor that the declarator name is an array name. The number of subscript expressions specified for the array indicates its dimensionality. The magnitude of the value given for the subscript expressions indicates the maximum value that the subscript may attain in any array element name.

No array element name may contain a subscript that, during execution of the executable program, assumes a value less than one or larger than the maximum length specified in the array declarator.

7.2.1.1.1. *Array Element Successor Function and Value of a Subscript.* For a given dimensionality, subscript declarator, and subscript, the value of a subscript pointing to an array element and the maximum value a subscript may attain are indicated in Table 2. A subscript expression must be greater than zero.

The value of the array element successor function is obtained by adding one to the entry in the subscript value column. Any array element whose subscript has this value is the successor to the original element. The last element of the array is the one whose subscript value is the maximum subscript value and has no successor element.

TABLE 2. VALUE OF A SUBSCRIPT

Dimensionality	*Subscript Declarator*	*Subscript*	*Subscript Value*	*Maximum Subscript Value*
1	(A)	(a)	a	A
2	(A, B)	(a, b)	$a + A \cdot (b - 1)$	$A \cdot B$

NOTES. 1. a and b are subscript expressions.
2. A and B are dimensions.

7.2.1.2. DIMENSION *Statement.* A DIMENSION statement is of the form:

$$\text{DIMENSION } v_1\ (i_1), v_2\ (i_2), \ldots, v_n\ (i_n)$$

where each $v\ (i)$ is an array declarator.

7.2.1.3. COMMON *Statement.* A COMMON statement is of the form:

$$\text{COMMON } a_1, a_2, \ldots, a_n$$

where each a is a variable name or an array name.

In any given COMMON statement, the entities occurring in the list of variable names are declared to be in common.

More than one COMMON statement may appear in a program unit. The processor will string together in common all entities so assigned in the order of their appearance. The first element of an array will follow the immediately preceding entity, if one exists, and the last element of an array will immediately precede the next entity if one exists.

The size of common in a program unit is the sum of the storage required for the

elements introduced through COMMON and EQUIVALENCE statements. The size of common in the various program units that are to be executed together need not be the same. Size is measured in terms of storage units (7.2.1.3.1).

7.2.1.3.1. *Correspondence of Common Blocks.* If all of the program units of an executable program that contain any definition of common define common such that there is identity in type for all entities defined in the corresponding position from the beginning of common, then the values in the corresponding positions are the same quantity in the executable program.

Each real or integer entity counts as one storage unit.

For common:

1. In all program units that have defined the identical type to a given position (counted by the number of preceding storage units) references to that position refer to the same quantity.

2. A correct reference is made to a particular position assuming a given type if the most recent value assignment to that position was of the same type.

7.2.1.4. EQUIVALENCE *Statement.* An EQUIVALENCE statement is of the form:

$$\text{EQUIVALENCE } (k_1), (k_2), \ldots, (k_n)$$

in which each k is a list of the form:

$$a_1, a_2, \ldots, a_m$$

Each a is either a variable name or an array element name (not a dummy argument), the subscript of which contains only constants, and m is greater than or equal to two. The number of subscript expressions of an array element name must correspond in number to the dimensionality of the array declarator or must be one (the array element successor function defines a relation by which an array can be made equivalent to a one-dimensional array of the same length).

The EQUIVALENCE statement is used to permit the sharing of storage by two or more entities. Each element in a given list is assigned the same storage (or part of the same storage) by the processor. The EQUIVALENCE statement should not be used to equate mathematically two or more entities.

The assignment of storage to variables and arrays declared directly in a COMMON statement is determined solely by consideration of their type and the COMMON and array declarator statements. Entities so declared are always assigned unique storage, contiguous in the order declared in the COMMON statement.

The effect of an EQUIVALENCE statement upon common assignment may be the lengthening of common; the only such lengthening permitted is that which extends common beyond the last assignment for common made directly by a COMMON statement.

When two variables or array elements share storage because of the effects of EQUIVALENCE statements the symbolic names of the variables or arrays in question may not both appear in COMMON statements in the same program unit.

Information contained in 7.2.1.1.1, 7.2.1.3.1, and the present section suffices to describe the possibilities of additional cases of sharing of storage between array elements and entities of common blocks. It is incorrect to cause either directly or indirectly a single storage unit to contain more than one element of the same array.

7.2.3. FORMAT *Statement.* FORMAT statements are used in conjunction with the input/output of formatted records to provide conversion and editing

information between the internal representation and the external character strings.

A FORMAT statement is of the form:

$$\text{FORMAT } (q_1t_1z_1t_2z_2 \ldots t_nz_nq_2)$$

where:

1. $(q_1t_1z_1t_2z_2 \ldots t_nz_nq_2)$ is the format specification.
2. Each q is a series of slashes or is empty.
3. Each t is a field descriptor or group of field descriptors.
4. Each z is a field separator.
5. n may be zero.

A FORMAT statement must be labelled.

7.2.3.1. *Field Descriptors.* The format field descriptors are of the forms:

$$r\text{F}w.d$$
$$r\text{E}w.d$$
$$r\text{I}w$$
$$n\text{H}h_1h_2 \ldots h_n$$
$$n\text{X}$$

where:

1. The letters F, E, I, H, and X indicate the manner of conversion and editing between the internal and external representations and are called the conversion codes.
2. w and n are nonzero integer constants representing the width of the field in the external character string.
3. d is an integer constant representing the number of digits in the fractional part of the external character string.
4. r, the repeat count, is an optional nonzero integer constant indicating the number of times to repeat the succeeding basic field descriptor.
5. Each h is one character.

For all descriptors, the field width must be specified. For descriptors of the form $w.d$, the d must be specified, even if it is zero. Further, w must be greater than or equal to d.

The phrase *basic field descriptor* will be used to signify the field descriptor unmodified by r.

The internal representation of external fields will correspond to the internal representation of the corresponding type constants (4.2 and 5.1.1).

7.2.3.2. *Field Separators.* The format field separators are the slash and the comma. A series of slashes is also a field separator. The field descriptors or groups of field descriptors are separated by a field separator.

The slash is used not only to separate field descriptors, but to specify demarcation of formatted records. A formatted record is a string of characters. The lengths of the strings for a given external medium are dependent upon both the processor and the external medium.

The processing of the number of characters that can be contained in a record by an external medium does not of itself cause the introduction or inception of processing of the next record.

7.2.3.3. *Repeat Specifications.* Repetition of the field descriptors (except nH and nX) is accomplished by using the repeat count. If the input/output list warrants, the specified conversion will be interpreted repetitively up to the specified number of times.

Repetition of a group of field descriptors or field separators is accomplished by enclosing them within parentheses and optionally preceding the left parenthesis with an integer constant called the group repeat count indicating the number of times to interpret the enclosed grouping. If no group repeat count is specified a group repeat count of one is assumed. This form of grouping is called a basic group.

7.2.3.4. *Format Control Interaction with an Input/Output List.* The inception of execution of a formatted READ or formatted WRITE statement initiates format control. Each action of format control depends on information jointly provided respectively by the next element of the input/output list, if one exists, and the next field descriptor obtained from the format specification. If there is an input/output list at least one field descriptor other than nH or nX must exist.

When a READ statement is executed under format control one record is read when the format control is initiated, and thereafter additional records are read only as the format specification demands. Such action may not require more characters of a record than it contains.

When a WRITE statement is executed under format control, writing of a record occurs each time the format specification demands that a new record be started. Termination of format control causes writing of the current record.

Except for the effects of repeat counts, the format specification is interpreted from left to right.

To each I, F, or E basic descriptor interpreted in a format specification, there corresponds one element specified by the input/output list. To each H or X basic descriptor there is no corresponding element specified by the input/output list, and the format control communicates information directly with the record. Whenever a slash is encountered, the format specification demands that a new record start or the preceding record terminate. During a READ operation, any unprocessed characters of the current record will be skipped at the time of termination of format control or when a slash is encountered.

Whenever the format control encounters an I, F, or E basic descriptor in a format specification, it determines if there is a corresponding element specified by the input/output list. If there is such an element, it transmits appropriately converted information between the element and the record and proceeds. If there is no corresponding element, the format control terminates.

If, however, the format control proceeds to the last outer right parenthesis of the format specification, a test is made to determine if another list element is specified. If not, control terminates. However, if another list element is specified the format control demands a new record start and control reverts to that group repeat specification terminated by the last preceding right parenthesis, or if none exists, then to the first left parenthesis of the format specification.

7.2.3.6. *Numeric Conversions.* The numeric field descriptors I, F, and E are used to specify input/output of integer and real data.

1. In numeric input fields blanks are permitted only to the left of the first nonblank character or between the sign of the field and the next nonblank character. Such blanks are treated as zero in conversion. Plus signs may be omitted. A field of all blanks is considered to be zero.

2. With the F and E input conversions, a decimal point appearing in the input field overrides the specification supplied by the field descriptor.

3. With all output conversions, the output field is right justified. If the number of characters produced by the conversion is smaller than the field width leading blanks will be inserted in the output field.

4. With all output conversions, the external representation of a negative value must be signed; a positive value may be signed.

5. The number of characters produced by an output conversion must not exceed the field width.

7.2.3.6.1. *Integer Conversion.* The numeric field descriptor Iw indicates that the external field occupies w positions as an integer. The value of the list item appears, or is to appear, internally as an integer datum.

In the external input field, the character string must be in the form of an integer constant or signed integer constant (5.1.1.1), except for the interpretation of blanks (7.2.3.6).

The external output field consists of blanks, if necessary, followed by a minus if the value of the internal datum is negative, or an optional plus otherwise, followed by the magnitude of the internal value converted to an integer constant.

7.2.3.6.2. *Real Conversions.* There are two conversions available for use with real data: F and E.

The numeric field descriptor F$w.d$ indicates that the external field occupies w positions, the fractional part of which consists of d digits. The value of the list item appears, or is to appear, internally as a real datum.

The external input field consists of an optional sign, followed by a string of digits optionally containing a decimal point.

The external output field consists of blanks, if necessary, followed by a minus if the internal value is negative, or an optional plus otherwise, followed by a string of digits containing a decimal point representing the magnitude, to d fractional digits, of the internal value.

The numeric field descriptor E$w.d$ indicates that the external field occupies w positions, the fractional part of which consists of d digits. The value of the list item appears, or is to appear, internally as a real datum.

The basic form of the external input field is the same as for the F conversion. The basic form may be followed by an exponent of one of the following forms:

1. Signed integer constant.
2. E followed by an integer constant.
3. E followed by a signed integer constant.

The standard form of the external output field is*

$$\xi 0.x_1 \ldots x_d Y$$

where:

1. $x_1 \ldots x_d$ are the d most significant digits of the value of the data to be output.
2. Y is of the form:

$$\mathrm{E} \pm y_1 y_2$$

and has the significance of a decimal exponent (an alternative for the plus in the first of these forms is the character blank).

3. Each y is a digit.

7.2.3.8. *Hollerith Field Descriptor.* Hollerith information may be transmitted by means of the field descriptor nH.

The nH descriptor causes Hollerith information to be read into, or written from,

* ξ signifies no character position or minus in that position.

the n characters (including blanks) following the nH descriptor in the format specification itself.

7.2.3.9. *Blank Field Descriptor.* The field descriptor for blanks is nX. On input, n characters of the external input record are skipped. On output, n blanks are inserted in the external output record.

8. PROCEDURES AND SUBPROGRAMS

There are four categories of procedures: statement functions, intrinsic functions, external functions, and external subroutines. The first three categories are referred to collectively as functions or function procedures; the last as subroutines or subroutine procedures. Function subprograms and subroutine subprograms are classified as procedure subprograms. Type rules for function procedures are given in 5.3.

8.1. STATEMENT FUNCTIONS. A statement function is defined internally to the program unit in which it is referenced. It is defined by a single statement similar in form to an arithmetic assignment statement.

In a given program unit all statement function definitions must precede the first executable statement of the program unit and must follow the specification statements, if any. The name of a statement function must not appear as a variable name or an array name in the same program unit.

8.1.1. *Defining Statement Functions.* A statement function is defined by a statement of the form:

$$f\ (a_1, a_2, \ldots, a_n) = e$$

where f is the function name, e is an expression, and the relationship between f and e must conform to the assignment rules in 7.1.1.1. The a's are distinct variable names, called the *dummy arguments* of the function. Since these are dummy arguments, their names, which serve only to indicate type, number, and order of arguments, may be the same as variable names of the same type appearing elsewhere in the program unit.

Aside from the dummy arguments, the expression e may only contain:

1. Constants.
2. Variable references.
3. Intrinsic function references.
4. References to previously defined statement functions.
5. External function references.

8.1.2. *Referencing Statement Functions.* A statement function is referenced by using its reference (5.2) as a primary in an arithmetic expression. The actual arguments, which constitute the argument list, must agree in order, number, and type with the corresponding dummy arguments. An actual argument in a statement function reference may be any expression of the same type as the corresponding dummy argument.

Execution of a statement function reference results in an association (10.2.2.) of actual argument values with the corresponding dummy arguments in the expression of the function definition, and an evaluation of the expression. Following this, the resultant value is made available to the expression that contained the function reference.

8.2. INTRINSIC FUNCTIONS AND THEIR REFERENCE. The symbolic names of the intrinsic functions (see Table 3) are predefined to the processor and have a special meaning and type if the name satisfies the conditions of 10.1.7.

An intrinsic function is referenced by using its reference as a primary in an arithmetic expression. The actual arguments, which constitute the argument list, must agree in type, number, and order with the specification in Table 3 and may be any expression of the specified type. The intrinsic functions SIGN and ISIGN are not defined when the value of the second argument is zero.

Execution of an intrinsic function reference results in the actions specified in Table 3 based on the values of the actual arguments. Following this, the resultant value is made available to the expression that contained the function reference.

8.3. EXTERNAL FUNCTIONS. As external function is defined externally to the program unit that references it. An external function defined by FORTRAN statements headed by a FUNCTION statement is called a function subprogram.

8.3.1. *Defining Function Subprograms.* A FUNCTION statement is of the form:

$$\text{FUNCTION } f\ (a_1, a_2, \ldots, a_n)$$

where:

1. f is the symbolic name of the function to be defined.
2. The a's, called the dummy arguments, are each either a variable name or an array name.

Function subprograms are constructed as specified in 9.1.3 with the following restrictions:

1. The symbolic name of the function must also appear as a variable name in the defining subprogram. During every execution of the subprogram this variable must be defined and, once defined, may be referenced or redefined. The value of the variable at the time of execution of any RETURN statement in this subprogram is called the value of the function.
2. The symbolic name of the function must not appear in any nonexecutable statement in this program unit, except as the symbolic name of the function in the FUNCTION statement.
3. The symbolic names of the dummy arguments may not appear in an EQUIVALENCE or COMMON statement in the function subprogram.
4. The function subprogram may not define or redefine any of its arguments nor any entity in common.
5. The function subprogram may contain any statements except SUBROUTINE, another FUNCTION statement, or any statement that directly or indirectly references the function being defined.
6. The function subprogram must contain at least one RETURN statement.

8.3.2. *Referencing External Functions.* An external function is referenced by using its reference (5.2) as a primary in an arithmetic expression. The actual arguments, which constitute the argument list, must agree in order, number, and type with the corresponding dummy arguments in the defining program unit. An actual argument in an external function reference may be one of the following:

1. A variable name.
2. An array element name.
3. An array name.
4. Any other expression.

Execution of an external function reference as described in the foregoing results in an association (10.2.2) of actual arguments with all appearances of dummy arguments in executable statements and function definition statements. If the actual argument is as specified in item (4) in the foregoing this association

is by value rather than by name. Following these associations, execution of the first executable statement of the defining subprogram is undertaken.

TABLE 3. INTRINSIC FUNCTIONS

Intrinsic Function	*Definition*	*Number of Arguments*	*Symbolic Name*	*Type of:*	
				Argument	*Function*
Absolute Value	$\lvert a \rvert$	1	ABS IABS	Real Integer	Real Integer
Float	Conversion from integer to real	1	FLOAT	Integer	Real
Fix	Conversion from real to integer	1	IFIX	Real	Integer
Transfer of Sign	Sign of a_2 times a_1	2	SIGN ISIGN	Real Integer	Real Integer

An actual argument that is an array element name containing variables in the subscript could in every case be replaced by the same array name with a constant subscript containing the same values as would be derived by computing the variable subscript just before the association of arguments took place.

If a dummy argument of an external function is an array name the corresponding actual argument must be an array name.

8.3.3. *Basic External Functions.* FORTRAN processors must supply the external functions listed in Table 4. Referencing of these functions is accomplished as described in 8.3.2. Arguments for which the result of these functions is not mathematically defined or is of type other than that specified are improper.

8.4. SUBROUTINE. An external subroutine is defined externally to the program unit that references it. An external subroutine defined by FORTRAN statements headed by a SUBROUTINE statement is called a subroutine subprogram.

TABLE 4. BASIC EXTERNAL FUNCTIONS

Basic External Function	*Definition*	*Number of Arguments*	*Symbolic Name*	*Type of:*	
				Argument	*Function*
Exponential	e^a	1	EXP	Real	Real
Natural logarithm	$\log_e(a)$	1	ALOG	Real	Real
Trigonometric sine	sine (a)	1	COS	Real	Real
Trigonometric cosine	cos (a)	1	COS	Real	Real
Hyperbolic tangent	tanh (a)	1	TANH	Real	Real
Square Root	$(a)^{1/2}$	1	SQRT	Real	Real
Arctangent	arctan(a)	1	ATAN	Real	Real

8.4.1. *Defining Subroutine Subprograms.* A SUBROUTINE statement is of one of the forms:

SUBROUTINE $s(a_1, a_2, \ldots, a_n)$
or
SUBROUTINE s

where:

1. s is the symbolic name of the subroutine to be defined.
2. The a's, called the dummy arguments, are each either a variable name or an array name.

Subroutine subprograms are constructed as specified in 9.1.3 with the following restrictions:

1. The symbolic name of the subroutine must not appear in any statement in this subprogram except as the symbolic name of the subroutine in the SUBROUTINE statement itself.
2. The symbolic names of the dummy arguments may not appear in an EQUIVALENCE or COMMON statement in the subprogram.
3. The subroutine subprogram may define or redefine one or more of its arguments so as to effectively return results.
4. The subroutine subprogram may contain any statements except FUNCTION, another SUBROUTINE statement, or any statement that directly or indirectly references the subroutine being defined.
5. The subroutine subprogram must contain at least one RETURN statement.

8.4.2. *Referencing Subroutines.* A subroutine is referenced by a CALL statement (7.1.2.4). The actual arguments, which constitute the argument list, must agree in order, number, and type with the corresponding dummy arguments in the defining program. An actual argument in a subroutine reference may be one of the following:

1. A variable name.
2. An array element name.
3. An array name.
4. Any other expression.

If an actual argument corresponds to a dummy argument that is defined or redefined in the referenced subprogram the actual argument must be a variable name, an array element name, or an array name.

Execution of a subroutine reference as described in the foregoing results in an association of actual arguments with all appearances of dummy arguments in executable statements or function definition statements. If the actual argument is as specified in item (4) in the foregoing this association is by value rather than by name. Following these associations, execution of the first executable statement of the defining subprogram is undertaken.

An actual argument that is an array element name containing variables in the subscript could, in every case, be replaced by the same array element name with a constant subscript containing the same values as would be derived by computing the variable subscript just before argument association took place.

If a dummy argument of an external function is an array name the corresponding actual argument must be an array name.

If a subroutine reference causes a dummy argument in the referenced subroutine to become associated with another dummy argument in the same sub-

routine or with an entity in common, a definition of either entity within the subroutine is prohibited.

9. PROGRAMS

An executable program is a collection of statements, comment lines, and end lines that completely (except for input data values and their effects) describe a computing procedure.

9.1. PROGRAM COMPONENTS. Programs consist of program parts, program bodies, and subprogram statements.

9.1.1. *Program Part.* A program part must contain at least one executable statement and may but need not contain FORMAT statements.

9.1.2. *Program Body.* A program body is a collection of optional specification statements optionally followed by statement function definitions, followed by a program part, followed by an end line. The specification statements must be in the following order: DIMENSION, COMMON, and EQUIVALENCE.

9.1.3. *Subprogram.* A subprogram consists of a SUBROUTINE or FUNCTION statement followed by a program body.

9.1.5. *Main Program.* A main program consists of a program body.

9.1.6. *Executable Program.* An executable program consists of a main program plus any number of subprograms, external procedures, or both.

9.1.7. *Program Unit.* A program unit is a main program or a subprogram.

9.2. NORMAL EXECUTION SEQUENCE. When an executable program begins operation execution commences with the execution of the first executable statement of the main program. A subprogram, when referenced, starts execution with execution of the first executable statement of that subprogram. Unless a statement is a GO TO, arithmetic IF, RETURN, or STOP statement or the terminal statement of a DO, completion of execution of that statement causes execution of the next following executable statement. The sequence of execution following execution of any of these statements is described in Section 7. A program part may not contain an executable statement that can never be executed.

A program part must contain a first executable statement.

10. INTRA- AND INTERPROGRAM RELATIONSHIPS

10.1 SYMBOLIC NAMES. A symbolic name has been defined to consist of from one to five alphanumeric characters, the first of which must be alphabetic. Sequences of characters that are format field descriptors or uniquely identify certain statement types, e.g. GO TO, READ, etc., are not symbolic names in such occurrences nor do they form the first characters of symbolic names in these cases. In a program unit, a symbolic name (perhaps qualified by a subscript) must identify an element of one (and usually only one) of the following classes:

Class I	An array and the elements of that array.
Class II	A variable.
Class III	A statement function.
Class IV	An intrinsic function.
Class V	An external function.
Class VI	A subroutine.

10.1.1. *Restrictions on Class.* In the program unit in which a symbolic name in Class V appears immediately following the word FUNCTION in a FUNCTION statement, that name must also be in Class II.

Once a symbolic name is used in Class V or VI in any unit of an executable program, no other program unit of that executable program may use that name to identify an entity of these classes other than the one originally identified.

In a program unit, no symbolic name can be in more than one class except as noted in the foregoing. There are no restrictions on uses of symbolic names in different program units of an executable program other than those noted in the foregoing.

10.1.2. *Implications of Mentions in Specification Statements.* A symbolic name is in Class I if it appears as a declarator name and is not in Class III. Only one such appearance for a symbolic name in a program unit is permitted.

A symbolic name that appears in a COMMON statement is either in Class I or in Class II but not Class V (8.3.1.) Only one such appearance for a symbolic name in a program unit is permitted.

A symbolic name that appears in an EQUIVALENCE statement is either in Class I or in Class II but not Class V (8.3.1).

10.1.3. *Array and Array Element.* In a program unit any appearance of a symbolic name that identifies an array must be immediately followed by a subscript, except for the following cases:

1. In the list of an input/output statement.
2. In a list of dummy arguments.
3. In the list of actual arguments in a reference to an external procedure.
4. In a COMMON statement.

Only when an actual argument of an external procedure reference is an array name may the corresponding dummy argument be an array name. If the actual argument is an array name the length of the dummy argument array must agree with the length of the actual argument array.

10.1.4. *External Procedures.* In the execution of an executable program, a procedure subprogram may not be referenced twice without the execution of a RETURN statement in that procedure having intervened.

10.1.5. *Subroutine.* A symbolic name is in Class VI if it appears:

1. Immediately following the word SUBROUTINE in a SUBROUTINE statement.
2. Immediately following the word CALL in a CALL statement.

10.1.6. *Statement Function.* A symbolic name is in Class III in a program unit if, and only if, it meets all three of the following conditions:

1. It is not in Class I or Class IV.
2. Every appearance of the name is immediately followed by a left parenthesis.
3. A function defining statement is present for that symbolic name.

10.1.7. *Intrinsic Function.* A symbolic name is in Class IV in a program unit if, and only if, it meets both of the following conditions:

1. The symbolic name appears in the name column of Table 3.
2. Every appearance of the symbolic name is immediately followed by an actual argument list enclosed in parentheses.

The use of an intrinsic function in a program unit of an executable program does not preclude the use of the same symbolic name to identify some other entity in a different program unit of that executable program.

10.1.8. *External Function.* A symbolic name is in Class V if it:

1. Appears immediately following the word FUNCTION in a FUNCTION statement.
2. Is not in Class I, Class III, Class IV, or Class VI and appears immediately followed by a left parenthesis on every occurrence. There must be at least one such appearance in the program unit in which it is so used.

10.1.9. *Variable.* In a program unit, a symbolic name is in Class II if it meets both of the following conditions:

1. It is not in Class VI.
2. It is never immediately followed by a left parenthesis unless it is immediately preceded by the word FUNCTION in a FUNCTION statement.

10.2. DEFINITION. There are two levels of definition of numeric values, first-level definition and second-level definition. The concept of definition on the first level applies to array elements and variables; that of second-level definition to integer variables only. These concepts are defined in terms of progression of execution; and thus, an executable program, complete and in execution, is assumed in what follows.

There is another variety of definition which refers to when an external procedure may be referenced, and it will be discussed in the next section.

In what follows, otherwise unqualified use of the terms definition and undefinition (or their alternate forms) as applied to variables and array elements will imply modification by the phrase "on the first level."

10.2.1. *Definition of Procedures.* If an executable program contains information describing an external procedure, such an external procedure with the applicable symbolic name is defined for use in that executable program. An external function reference or subroutine reference (as the case may be) to that symbolic name may then appear in the executable program, provided that number of arguments agrees between definition and reference. Other restrictions on agreements are contained in 8.3.1, 8.3.2, 8.4.1, 8.4.2, and 10.1.3.

The basic external functions listed in Section 8.3.3 are always defined and may be referenced subject to the restrictions alluded to in the foregoing.

A symbolic name in Class III or Class IV is defined for such use.

10.2.2. *Associations That Effect Definition.* Entities may become associated by:

1. COMMON association.
2. EQUIVALENCE association.
3. Argument substitution.

Multiple association to one or more entities can be the result of combinations of the foregoing. Any definition or undefinition of one of a set of associated entities affects the definition or undefinition of each entity of the entire set.

For purposes of definition, in a program unit there is no association between any two entities both of which appear in COMMON statements. Further, there is no other association for common and equivalenced entities other than those stated in 7.2.1.3.1 and 7.2.1.4.

If an actual argument of an external procedure reference is an array name, an array element name, or a variable name, then the discussions in 10.1.3 and 10.2.1 allow an association of dummy arguments with the actual arguments only between the time of execution of the first executable statement of the procedure and the inception of execution of the next encountered RETURN statement of that

procedure. Note specifically that this association can be carried through more than one level of external procedure reference.

In what follows, variables or array elements associated by the information in 7.2.1.3.1 and 7.2.1.4 will be equivalent if, and only if, they are of the same type.

If an entity of a given type becomes defined, then all associated entities of different type become undefined at the same time, while all associated entities of the same type become defined unless otherwise noted.

Association by argument substitution is valid only in the case of identity of type so the rule in this case is that an entity created by argument substitution is defined at time of entry if, and only if, the actual argument was defined. If an entity created by argument substitution becomes defined or undefined (while the association exists) during execution of a subprogram, then the corresponding actual entities in all calling program units become defined or undefined accordingly.

10.2.3. *Events that Effect Definition.* Any entity is undefined at the time of the first execution of the first executable statement of the main program. Redefinition of a defined entity is always permissible except for certain integer variables (7.1.2.8 and 7.1.3.1.1) or certain entities in subprograms (6.4, 8.3.2, and 8.4.2).

Variables and array elements become defined or redefined as follows:

1. Completion of execution of an arithmetic assignment statement causes definition of the entity which precedes the equals.
2. As execution of an input statement proceeds, each entity, which is assigned a value of its corresponding type from the input medium, is defined at the time of such association and associated entities become undefined. Only at the completion of execution of the statement do associated entities of the same type become defined.
3. Completion of execution of a DO statement causes definition of the control variable.
4. Inception of execution of action specified by a DO-implied list causes definition of the control variable.

Variables and array elements become undefined as follows:

1. At the time a DO is satisfied, the control variable becomes undefined.
2. Completion of execution of action specified by a DO-implied list causes undefinition of the control variable.
3. When an associated entity of different type becomes defined.
4. When an associated entity of the same type becomes undefined.

10.2.6. *Entities Not in Common.* An entity not in common is initially undefined. Such entities once defined by any of the rules previously mentioned, remain defined until they become undefined.

If such an entity is in a subprogram, the completion of execution of a RETURN statement in that subprogram causes all such entities and their associates at that time to become undefined. In this respect, it should be noted that the association between dummy arguments and actual arguments is terminated at the inception of execution of the RETURN statement.

10.2.7. *Basic Block.* In a program unit, a basic block is a group of one or more executable statements defined as follows.

The following statements are block terminal statements:

1. DO statement.
2. CALL statement.

3. GO TO statement of all types.
4. Arithmetic IF statement.
5. STOP statement.
6. RETURN statement.
7. The first executable statement, if it exists, preceding a statement whose label is mentioned in a GO TO or arithmetic IF statement.
8. An arithmetic statement in which an integer variable precedes the equals.
9. A READ statement with an integer variable in the list.

The following statements are block initial statements:

1. The first executable statement of a program unit.
2. The first executable statement, if it exists, following a block terminal statement.

Every block initial statement defines a basic block. If that initial statement is also a block terminal statement the basic block consists of that one statement. Otherwise, the basic block consists of the initial statement and all executable statements that follow until a block terminal statement is encountered. The terminal statement is included in the basic block.

10.2.7.1. *Last Executable Statement.* In a program unit the last executable statement must be one of the following statements: GO TO statement, arithmetic IF statement, STOP statement, or RETURN statement.

10.2.8. *Second Level Definition.* Integer variables must be defined on the second level when used in subscripts and computed GO TO statements.

Redefinition of an integer entity causes all associated variables to be undefined on the second level during the execution of the program until the associated integer variable is explicitly redefined.

Except as just noted, an integer variable is defined on the second level upon execution of the initial statement of a basic block only if both of the following conditions apply:

1. The variable is used in a subscript or in a computed GO TO in the basic block in question.
2. The variable is defined on the first level at the time of execution of the initial statement in question.

This definition persists until one of the following happens:

1. Completion of execution of the terminal statement of the basic block in question.
2. The variable in question becomes undefined or receives a new definition on the first level.

At this time, the variable becomes undefined on the second level.

In addition, the occurrence of an integer variable in the list of an input statement in which that integer variable appears following in a subscript causes that variable to be defined on the second level. This definition persists until one of the following happens:

1. Completion of execution of the terminal statement of the basic block containing the input statement.
2. The variable becomes undefined or receives a new definition on the first level.

An integer variable defined as the control variable of a DO-implied list is defined on the second level over the range of that DO-implied list and only over that range.

10.2.9. *Certain Entities in Function Subprograms.* If a function subprogram is referenced more than once with an identical argument list in a single statement, the execution of that subprogram must yield identical results for those cases mentioned, no matter what the order of evaluation of the statement.

10.3. DEFINITION REQUIREMENTS FOR USE OF ENTITIES. Any variable referenced in a subscript or a computed GO TO must be defined on the second level at the time of this use.

Any variable, array element, or function referenced as a primary in an expression and any subroutine referenced by a CALL statement must be defined at the time of this use. In the case where an actual argument in the argument list of an external procedure reference is a variable name or an array element name, this in itself is not a requirement that the entity be defined at the time of the procedure reference.

Any variable used as an initial value, terminal value, or incrementation value of a DO statement or a DO-implied list must be defined at the time of this use.

Any variable used to identify an input/output unit must be defined at the time of this use.

At the time of execution of a RETURN statement in a function subprogram, the value of that function must be defined.

At the time of execution of an output statement, every entity whose value is to be transferred to the output medium must be defined. If the output is under control of a format specification, a correct association of conversion code with type of entity is required. The following are the correct associations: I with integer; and E and F with real.