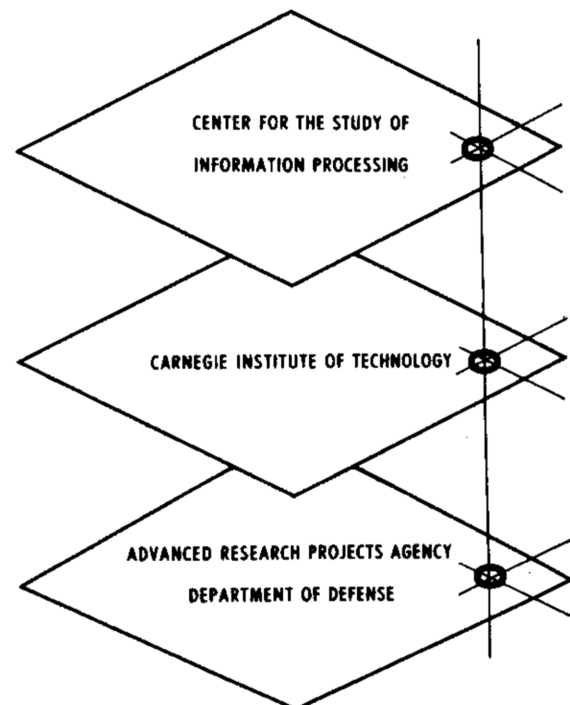
ALGOL - 20

A LANGUAGE MANUAL

JANET W. FIERST, EDITOR
DAVID M. BLOCHER
ROBERT T. BRADEN
ARTHUR EVANS JR.
RICHARD B. GROVE



ALGOL-20

A LANGUAGE MANUAL

JANET W. FIERST, EDITOR
DAVID M. BLOCHER
ROBERT T. BRADEN
ARTHUR EVANS JR.
RICHARD B. GROVE

FIRST PRINTING FEBRUARY 1965

THIS WORK WAS SUPPORTED BY THE
ADVANCED RESEARCH PROJECTS AGENCY OF THE
OFFICE OF THE SECRETARY OF DEFENSE:
CONTRACT SD-146

CARNEGIE INSTITUTE OF TECHNOLOGY

Acknowledgements

The construction of the programming system described here is the result of the combined effort of many people. The following were involved with coding the translator: David M. Blocher, Arthur Evans, Jr., Janet W. Fierst, Richard B. Grove and Carol H. Thompson. Ronald R. Bushyager, III, wrote WHAT, the assembly language processor included in the translator. Charles L. Thornton wrote a table loader (the "Meta-compiler") which is an essential part of the process of assembling the translator. Grove wrote the relocator and the librarian. Special thanks are owed to Robert T. Braden for acting as the conscience of the group with many useful suggestions on "ALGOL esthetics". The entire task was directed by Evans.

This document has been edited by Fierst, who also did much of the writing. The following people, in addition, contributed to the writing and editing of the document: Blocher, Braden, Evans and Grove. Ronald P. Hackleman wrote many of the relocatable library routine descriptions appearing in Chapter 5. The typing has been done by Edythe Simmons, and Robert D. Smith contributed materially with his editorial assistance.

PREFACE

ALGOL-20 is a realization of the international language ALGOL-60. The international language, although a valuable vehicle for the description of algorithms, does not really become useful until it is implemented on computers. However, each implementer has found it necessary in some cases and desirable in others to make changes in the language. Further, additions to the language such as input/output are necessary. This ALGOL-20 Manual is a description of the realization of ALGOL as implemented at Carnegie Institute of Technology.

Two additional documents are needed to complete the description of Carnegie Tech ALGOL. One is a description of ALIBN - the librarian used in connection with the two libraries. The other document describes the assembly language - WHAT - which is built into the Algol translator. The user may include assembly code as part of his program, as described in the WHAT manual. These manuals are currently in preparation.

The internal operation of the translator has not been adequately described. However, An ALGOL 60 Compiler, by A. Evans, which was printed in Annual Review of Automatic Programming, Volume 4, Pergamon Press, describes part of the translator. A preliminary version of the format language used was described in A Format Language, by Alan J. Perlis, in Comm. ACM, 7(Feb. 1964), pp. 89-96.

Arthur Evans, Jr.
January, 1965

Introduction

The manual is organized as follows: Chapter 0 is a ready reference containing in summary form the information the experienced programmer will need. It is not suitable for reading by itself, but is useful for reference to particular points. Chapter 1 is an introduction which includes bibliographical citations to several introductory texts on ALGOL, for the programmer who does not yet know the language. Chapter 2 describes in considerable detail how the local system differs from the international language. Chapter 3 contains a detailed description of the input/output system provided at Carnegie Tech. A format language of some sophistication is defined. Chapter 4 contains a description of system statements - those statements used to communicate to the translator information which is not part of the ALGOL language. Chapter 5 contains a description of the two libraries available to the translator and contains descriptions of the routines currently in the libraries. Chapter 6 is a collection of miscellaneous topics, including keypunch conventions, error codes, etc. Chapter 7 includes the ALGOL-60 report as revised in 1962 and a summarized list of differences between ALGOL-60 and local ALGOL.

Page numbers are of the form AL.m.n, where m indicates the chapter number and n is the page in the chapter. Two chapters -- three and six -- are divided into sub-chapters distinguished by lower case letters, e.g., Chapter 3b. The sub-chapters are also paged individually so that the first page of Chapter 3b is AL.3b.1, immediately following AL.3a.2.

CONTENTS

CHAPTER 0	ALGOL Ready Reference	0.1 - 0.16
1	Introduction	1.1 - 1.2
2	Notes on ALGOL at Carnegie Tech	2.1 - 2.14
3	ALGOL-20 Input/Output	3.1 - 3
	a. Introduction	
	b. Primer on PRINT	
	c. Primer on READ	
	d. Complete Description of All I/O Commands	
4.	System Statements	4.1 - 4.6
5.	The ALGOL Library	5.1 - 5.3
	a. Introduction	
	b. Routines in the Library	
6.	Miscellaneous	
	a. ALGOL-20 Card Format and Keypunching Conventions	6a.1 - 6a.2
	b. ALGOL-20 Error Messages	6b.1 - 6b.7
	c. Printing of the Compiled Program	6c.1
	d. Privileged Identifiers	6d.1 - 6d.4
	e. Machine-Dependent Features Octal Constants String Constants Logic Variables Half Variables Index Variables	6e.1 - 6e.4
	f. Segments	6f.1
	g. Disc/Tape Routines	
	h. Storage Allocation	6h.1 - 6h.2
7.	ALGOL-60	
	a. The Revised ALGOL-60 Report	7a.1 - 7a.17
	b. Features of ALGOL-60 Which are Changed in ALGOL-20	7b.1 - 7b.2
	c. Restrictions on ALGOL-20 to Transform it into a Subset of ALGOL-60	7c.1 - 7c.2

Chapter 0 - ALGOL READY REFERENCE

ALGOL Notes and Error Messages 2

- Compile Errors 2
- Notes 5
- Run Errors 5

Keypunching 6

Precedence Rules for Operators and Relations 6

Format Instructions 7

- Control Instructions 7
- Instructions for PRINT and PUNCH 7
- Instructions for READ 9

Library Routines and Standard Functions 11

- Relocatable Routines 11
- Symbolic Routines 11
- Standard Functions 12

Reserved Identifiers 13

Privileged Identifiers 13

ALGOL READY REFERENCE

ALGOL Notes and Error Messages (Chapter 6b)

Compile Errors

Phase I Errors (Each of these errors terminates Phase II.)

0: The program does not start with begin.
1: A statement starts with an illegal character or an illegal reserved word.
2: A statement starts with an identifier followed by an illegal character.
3: In an expression an operand was expected and was not found.
4: In an expression a binary operator was expected and was not found. (Possibly caused by a semicolon missing after the preceding statement.)
5: A "]" does not have a matching "[".
6: An array element has been used illegally.
7: A ":" has appeared incorrectly.
8: A "←" or ":=" has appeared incorrectly.
9: A ")" does not have a matching "(".
10: A "," has appeared incorrectly.
11: then has appeared without if.
12: else has appeared without then.
13: Characters are still in the stack after a ";" or an end.
14: A procedure statement is followed by other than end, else, or ";".
15: for is not followed by an identifier.
16: The for variable is not followed by a "←" or ":=".
17: step has appeared without for.
18: until has appeared without step.
19: while has appeared without for.
20: do has appeared without for.
21: go to is not followed by an identifier or "(" or if.
22: go to if...then...is not followed by else.
23:
24: An obscure error in a go to statement.
25: An impossible error after begin. ("⊢" is not the second element in the stack. See Error 98.)
26: own is followed by something other than <type>.
27: An array declaration does not specify subscript bounds.
28: The identifier list of a declaration is not followed by a ";".
29: switch is not followed by an identifier.
30: The identifier of a switch declaration is not followed by a "←" or ":=".
31:
32: procedure is not followed by an identifier.
33: A procedure identifier is not followed by "(" or ";".
34: A formal parameter list is not followed by ")".
35: The ")" following a parameter list is not followed by a ";".
36: The identifier list in a specification is not followed by a ";".
37: An identifier did not follow the "," in an identifier list.
38: The illegal construction "then if" has occurred.
39: A switch with more than one subscript position has been used.
40: The value part of a procedure declaration was not followed by a ";".

ALGOL READY REFERENCE

41: The name of a permanent subroutine (such as "SIN") is not followed by "(".
42: There is an extra "," or else a missing ":" in an array declaration.
43: More begin's than end's have occurred when the end-of-file is reached.
44: Impossible - see Error 98.
45: max or min is not followed by "(".
46: In an array declaration the identifier list is not followed by "[".
47: Array specifier has subscript bounds, which it should not.
48: library is not followed by <type> or procedure.
49:

Phase I Errors (format and name statements) (Each of these errors terminates Phase II.)

50: A reserved input/output word is not followed by "(" .
51: A format list element starts with an illegal character. (Should be "<" or "→" or "$" or identifier).
52: "→" is missing: i.e., a replicator was expected but not found.
53: for is missing after "$".
54: "→" is not followed by "$" or an identifier.
55: ")" or ">" is not followed by ")" or ",".
56: A name statement or format statement is not followed by end, else or ";".
57: A replicator is not followed by "(" or "<".
58: "<" or "," is followed by an illegal character.
59: An integer is followed by an illegal character.
60: A format instruction is not followed by ">" or ",".
61: An illegal prefix to a numeric primary has been used.
62: An illegal numeric primary has been used.
63: "." appears in a numeric primary in a read statement.
64: In a numeric primary, E, F or S is not followed by an integer.
65:
66:
67:
68:
69:

Phase II Errors (Only those errors marked "*" turn off Phase II.)

*70: A reserved word which is not yet available has been used.
71: A label has been used but not defined. (The name of the label is printed prior to this error message)
72: An identifier has been used but not declared.
73: An identifier has been declared twice in this block.
74: An identifier in the value list is not a parameter.
75: An identifier which has been used as a procedure has not been declared to be one.
76: A subscripted identifier has not been declared to be an array or switch.
77: The program is too long.
78: A procedure identifier which is not a function designator has been used in an expression.

* Turns off Phase II.

ALGOL READY REFERENCE

79: An identifier which has been used as a switch has not been declared to be one.
80: An array identifier has been used without subscripts.
81: Too many index variables have been declared.
82: A label or array or switch has been called by value.
83: An identifier in a specification list is not a parameter.
84: In a procedure declaration a parameter is not specified.
85: In a procedure declaration a parameter is specified twice.
86: A procedure, switch or label appears on the left of a ":=" or "←".
*87: The W2 stack is too full.
88: More than 100 relocatable <u>library</u> <u>procedures</u> have been declared.
89: A constant has been used in place of an identifier, e.g., 33[k].
*90: A subscripted for variable has been used (this is not yet available in ALGOL-20).
*91: The next-command pointer is less than the base of the program.
92:
93:
94:
95:

Miscellaneous Errors

96:
97: A possible translator error - bring listing to Janet Fierst at the Computation Center.
98: Impossible: bring your listing to A. Evans at the Computation Center.
99: Same as 98.

Subscan Errors

100: A card column contains an illegal combination of punches.
101: Too many abcons or adcons have been used (numerical constants and alphanumeric string constants).
102: Too many decimal points appear in a number.
103: Too many "$_{10}$"s appear in a number.
104: An error has appeared in a parameter delimiter comment: ")<any string not containing:>:(".
105: An illegal bar ("|") variable has been used.
106: A constant has been used which is too large to fit into a <u>real</u> variable.
107: A "$_{10}$" is followed by something other than "+", "-", or <digit>.
108: A string goes over the end of a card.
*109: The symbol table has been exceeded.
110:
111:
112:
113:
114:

* Turns off Phase II.

ALGOL READY REFERENCE

System Statement Errors

115: An abcon system statement has occurred after code has been compiled.
116:
117: An abcon system statement has requested more space than there is in user memory.
118:
119: An illegal SY card has occurred. (This may be caused by a LIBRARY card after the symbolic library has been released.)
120: The library procedure nesting exceeds 5.
121:
122: WHAT has been called after it has been released.
123: An illegal segment statement has been used.
124: An SY LIBRARY card has asked for a routine not in the symbolic library.
125: A library procedure declaration has named a routine not in the relocatable library.

Notes

Note 1: end comment convention was used on preceding card. That is, everything was ignored up to ";", end, or else.
Note 2: A function designator has been used as a procedure statement.
Note 3: In an arithmetic or boolean expression, the construction if...then if has occurred. This is syntactically illegal but unambiguous, and is therefore accepted by the translator.
Note 4: An arithmetic (boolean) (designational) expression has been used where a simple arithmetic (boolean) (designational) expression should have been used.
Note 5: In a designational expression, the construction if...then if has occurred. This is syntactically illegal but unambiguous.
Note 6: Phase II has been turned off.
Note 7: The construction if...then for...do...else... which is legal in ALGOL 60 but illegal in ALGOL 62 has been used.
Note 8: TAB appears as a character.
Note 9: Fifty errors have been found on a single card; compilation has been terminated.

Run Errors

ADRP	address--opcode fault
CFLG	command flag error
EXP	EXP (x) called with $X > 160.116998$
EXPO	exponent overflow
LN	LN (x) called with $X \leq 0$
RAD1	upper < lower in a bound pair in an array declaration
RAD2	declared arrays exceed available space
READ	an error has occurred in reading a data card

ALGOL READY REFERENCE

SIN	the argument to SIN or COS is greater than 8↑21.
SQRT	SQRT (X) called with X < 0
TIMR	time limit exceeded
X↑A1	X = 0 and A ≤ 0
X↑A2	A * LN (X) > 160.116998
X↑A3	X < 0 and A not integer valued

Keypunching (Chapter 6a)

```
                          111 1 1 111 11 2 222 2222223333333333...R
column  -   12 3 45678 9012 3 4 567 89 0 123 4567890123456789   M
WHAT        WH  .LOC.       F   OP.    M      Addr,Index;comment..
ALGOL       AL  ...................ALGOL text.................
system      SY  ...................system text................
comment     CO  .....................comment..................
```

RM is the right margin. (Initially RM = 72. It may be changed by SY RIGHT MARGIN - see Chapter 4.)

Precedence rules for operators and relations (Chapter 2)

↓ (performed first)

MOD

↑

+ - (unary)

* /

+ - (binary)

< > ¬< ¬> = ≠

¬

∧

∨ (performed last)

ALGOL READY REFERENCE

Format Instructions (Chapter 3)

Control Instructions

nC	Sets CP to column n (CP ← n).
nR	Moves CP n columns to the right (CP ← CP + n).
nL	Moves CP n columns to the left (CP ← CP - n).

Instructions for PRINT and PUNCH

Control and Alphanumerics

nE	Prints or punches the contents of the appropriate buffer, clears the buffer to blanks, prints or punches n-1 blank lines, and sets CP to left margin.
nW	Prints or punches the contents of the appropriate buffer n times. CP and the buffer are not changed.
P	Upspaces the paper to a new page and prints a page header on the first line. CP and the buffer are not changed. This instruction is ignored in PUNCH.
'<string>'	Stores the characters of the string into the appropriate buffer.
nB	Stores n blanks.
nQ	Stores n quotes (').
nA	Stores n characters which are taken from ↓ ((n+3)/4) names.
nT	Stores min (5, n) characters which are taken from one of the internal strings 'TRUE ', 'FALSE' or 'UNDEF', according to the boolean value of the corresponding name.

Numeric Instructions

1. Prefix

L$	Stores a dollar sign left-justified.
$	Stores a dollar sign immediately before the first digit.
L+(L-)	Stores the sign (sign if minus) left-justified or immediately after a '$' stored by "L$".

ALGOL READY REFERENCE

+(-)	Stores the sign (sign if minus) immediately before the first digit or a '$' stored by "$". If the number is negative, a minus sign is stored; if the number is non-negative, a plus sign (blank) is stored.

2. Numeric Primary

nD	Stores n digits or blanks. Leading (or trailing) zeros are replaced by blanks.
nZ	Stores n digits.
	Stores a decimal point, '.'

3. Suffix

L	Shifts the number until the left-most digit is non-zero (if possible) and stores the resultant exponent.
E(F)$\pm$n	Shifts the number until its exponent equals $\pm$ n. The resultant exponent is (is not) printed.
S$\pm$n	Shifts the number until the left-most digit is non-zero (if possible). The decimal point is inserted in the position to give an exponent equal to $\pm$n and the exponent is stored. The numeric primary must be of the form "mD." or "mZ.".
H	Converts and stores the number in octal (base 8) instead of decimal.
K	Invokes special spacing. If the prefix contains "L$" or "$", the digits of the number are stored in groups of three separated by commas. If neither "L$" nor "$" appears, the digits are stored in groups of five, separated by blanks.
N	Suppresses error printing which may occur when left-most non-zero digits overflow the field specified by the numeric primary.
T	Truncates the number at the last digit stored. Normally, numbers are rounded by adding five to the first digit not printed.

ALGOL READY REFERENCE

Instructions for READ

Control and Alphanumeric Instructions

nE	Reads n card images into the READ buffer and sets CP to left margin. Only the last card image read is available after the instruction is executed.
nW	Functions as "nE" except that the card images are also listed on the printer.
'<string>'	Causes the n characters of the string to be stored, four per word, into the next $\downarrow((n+3)/4)$ names. If n is not a multiple of four, characters in the last name are stored right-justified. The CP and buffer are not changed.
nA	Scans the characters in the next n positions of the buffer and stores them as in the '<string>' instruction.
nT	Scans the characters in the next n positions of the buffer. If the first non-blank character is the letter "T", the value <u>true</u> is stored in the next name; otherwise, the value of the name is set to <u>false</u>. The corresponding name must be of type <u>boolean</u> or <u>logic</u>.

Numeric Instructions

1. Numeric Primary

nD	Scans the number represented in the next n positions of the buffer. Blanks are ignored. Numbers are in free field format and may contain signs, decimal points, and exponents. Numbers preceded by a "/" are treated as octal quantities.
nZ	Scans as "nD" except that blanks are treated as zeros.

2. Suffix

H	Assumes the number read to be octal (base 8) instead of decimal.

ALGOL READY REFERENCE

E±n — Multiplies the number read by ten (eight) raised to the power ±n.

N — Causes illegal characters (such as letters) to be ignored.

3. Free Read

nF — Scans and concatenates n octal or decimal numbers in fields separated by commas. Blanks are ignored with the exception that if an entire field is blank, the corresponding name is unchanged. Numbers preceded by a "/" are treated as octal. A number field may be continued over the end of a card image. The last number in the data group must be followed by either a "," or a "*".

ALGOL READY REFERENCE

Library Routines and Standard Functions (Chapter 5)

Relocatable Routines (i.e., library procedures)

AND.CALL	sets the scratch pointer and enters AND
AND.FILE	assigns a logical file type to an AND file
DISC.READ	reads from disc or tape
DISC.WRITE	writes on disc or tape
GOOF.STAR	prints a run-error message
GO.SEG	slews to a segment
LINK	links to a segment
RUN.ERROR	sets up run-error recovery
SLEW	slews to a record
SYSTEM.DUMP	dumps an ALGOL program as a system

Symbolic Routines

AND.PUNCH	punches an AND record onto cards
BANSOLV	solves a system of linear equations whose coefficient matrix is a band matrix
COMDIV	computes the quotient of two complex numbers
CURVEFIT	determines the best least squares polynomial approximation to a given curve, with or without constraints
CURFIT	determines the best least squares polynomial approximation to a given curve, with constraints
ELIPS	computes the values of the complete elliptic integrals of the first and second kinds
FREQ	determines the frequency distribution of a given set of data
GAME	solves a finite, zero-sum, two-person game
GJR	computes the inverse of a given matrix
HERMJA	finds all the eigenvectors and eigenvalues of a given Hermitian matrix
JACOBI	finds all eigenvectors and eigenvalues of a given symmetric matrix
MULLER	finds the real and complex roots of a general equation of the form $f(z) = 0$

ALGOL READY REFERENCE

NEVILLE	computes approximate values of a tabulated function by interpolation
NORMRAN	computes a sequence of normally distributed pseudo-random numbers
PLOT	produces the graph of one to ten functions
RANDOM	computes a sequence of uniformly distributed pseudo-random numbers
SIM	performs numerical integration by Simpson's method
SORT1	sorts a list of numbers into ascending order

Standard Functions (i.e., built-in functions)

ABS	absolute value
ARCTAN	arctangent
BUFFERSET	redefines the input/output buffers
CLOCK	time since the last job-card minus parameter (in seconds)
COS	cosine
DEBUGPRINT	a fixed format print routine
ENTIER	the largest integer which is not greater than the parameter
EXP	exponential (e^x)
HALT	halt
LN	natural logarithm
MAX	maximum } (see page AL.2.10 in the ALGOL manual)
MIN	minimum }
PAGES	number of pages since the jobcard
PAUSE	saves the program for restart
PRINT	controls printing
SIGN	-1 if parameter negative, +1 if positive, 0 if zero
SIN	sine
SQRT	square root
TIME	time since midnight (in seconds)

MOD is an operator such that for integer m and n the quantity m MOD n is the remainder on dividing m by n.

ALGOL READY REFERENCE

Library Routines and Standard Functions (Chapter 5)

Relocatable Routines (i.e., library procedures)

AND.CALL	sets the scratch pointer and enters AND
AND.FILE	assigns a logical file type to an AND file
DISC.READ	reads from disc or tape
DISC.WRITE	writes on disc or tape
GOOF.STAR	prints a run-error message
GO.SEG	slews to a segment
LINK	links to a segment
RUN.ERROR	sets up run-error recovery
SLEW	slews to a record
SYSTEM.DUMP	dumps an ALGOL program as a system

Symbolic Routines

AND.PUNCH	punches an AND record onto cards
BANSOLV	solves a system of linear equations whose coefficient matrix is a band matrix
COMDIV	computes the quotient of two complex numbers
CURVEFIT	determines the best least squares polynomial approximation to a given curve, with or without constraints
CURFIT	determines the best least squares polynomial approximation to a given curve, with constraints
ELIPS	computes the values of the complete elliptic integrals of the first and second kinds
FREQ	determines the frequency distribution of a given set of data
GAME	solves a finite, zero-sum, two-person game
GJR	computes the inverse of a given matrix
HERMJA	finds all the eigenvectors and eigenvalues of a given Hermitian matrix
JACOBI	finds all eigenvectors and eigenvalues of a given symmetric matrix
MULLER	finds the real and complex roots of a general equation of the form $f(z) = 0$

ALGOL READY REFERENCE

NEVILLE	computes approximate values of a tabulated function by interpolation
NORMRAN	computes a sequence of normally distributed pseudo-random numbers
PLOT	produces the graph of one to ten functions
RANDOM	computes a sequence of uniformly distributed pseudo-random numbers
SIM	performs numerical integration by Simpson's method
SORT1	sorts a list of numbers into ascending order

Standard Functions (i.e., built-in functions)

ABS	absolute value
ARCTAN	arctangent
BUFFERSET	redefines the input/output buffers
CLOCK	time since the last job-card minus parameter (in seconds)
COS	cosine
DEBUGPRINT	a fixed format print routine
ENTIER	the largest integer which is not greater than the parameter
EXP	exponential (e^x)
HALT	halt
LN	natural logarithm
MAX	maximum } (see page AL.2.10 in the ALGOL manual)
MIN	minimum }
PAGES	number of pages since the jobcard
PAUSE	saves the program for restart
PRINT	controls printing
SIGN	-1 if parameter negative, +1 if positive, 0 if zero
SIN	sine
SQRT	square root
TIME	time since midnight (in seconds)

MOD is an operator such that for integer m and n the quantity m MOD n is the remainder on dividing m by n.

ALGOL READY REFERENCE

Reserved Identifiers (Chapter 2)

ABS	(2)	GO TO	(1)	PRINT	(3-3.1ff)
ARCTAN	(2)	HALF	(3-2.5)	PROCEDURE	(1)
ARRAY	(1)	IF	(1)	PUNCH	(3-3.1ff)
BEGIN	(1)	INDEX	(3-2.5)	READ	(3-3.1ff)
BOOLEAN	(1)	INPUT	(3-NA)	REAL	(1)
COMMENT	(1)	INTEGER	(1)	SIGN	(2)
COS	(2)	LABEL	(1, 3-2.10)	SIN	(2)
DO	(1)	LIBRARY	(3-5.1ff)	SQRT	(2)
ELSE	(1)	LN	(2)	STEP	(1)
END	(1)	LOGIC	(3-2.5)	STRING	(1)
ENTIER	(2)	MAX	(3-2.9)	SWITCH	(1)
EXP	(2)	MIN	(3-2.9)	THEN	(1)
FALSE	(1)	MOD	(3-2.9)	TRUE	(1)
FOR	(1)	MONITOR	(3-NA)	UNTIL	(1)
FORWARD	(3-NA)	NAME	(3-3.1ff)	VALUE	(1)
GO	(1)	OUTPUT	(3-NA)	WHILE	(1)
GOTO	(1)	OWN	(1)		

(1) ALGOL 60 "built in" word.

(2) ALGOL 60 reserved function identifier.

(3) ALGOL 20 reserved word--see page reference. (NA means not now available.)

Privileged Identifiers (Chapter 6d)

ACC	real	Accumulator
CLOCK	integer procedure	(time in seconds since job-card) minus parameter
DAY	logic	'␣dd␣' dd = day of month
DEBUGPRINT	procedure	fixed format print routine
EPSILON	real	smallest positive number = 8↑-63
HALT	procedure	halt
INFINITY	real	largest positive number = (8↑14-1)*8↑63
MONTH	logic	'mmm␣' mmm = name of month
PAGES	integer procedure	number of pages since job-card
PAUSE	procedure	save for restart
PRINT	procedure	controls printing on teletype
TIME	integer procedure	time in seconds since midnight
YEAR	logic	'yy␣␣' yy = last two digits of year

¬ labels in WHAT and ALIBN

Label	
¬ 0	Complement 0 when accessed arithmetically
¬ 1	1 flag
¬ 2	2 flag
¬ 3	3 flag
¬ 4	Function variable for relocatable library procedures
¬ 5	INFINITY = (8↑14-1) * 8↑13 (Chapter 6d)
¬ 6	EPSILON = 8↑(-63) (Chapter 6d)
¬ 7	Dynamic block level - pointer to array stack (an index register)
¬ 8	
¬ 9	
¬ 10	Exit from FORMAT and NAME
¬ 11	Current NAME list
¬ 12	Current FORMAT list
¬ 13	NAME routine
¬ 14	PRINT routine
¬ 15	PUNCH routine
¬ 16	READ routine
¬ 17	FORMAT routine
¬ 18	Page-and-print-page-header routine
¬ 19	
¬ 20	go to <label> routine
¬ 21	RAD - run-time array declaration routine
¬ 22	begin administration routine
¬ 23	end administration routine
¬ 24	procedure begin administration routine
¬ 25	procedure end administration routine
¬ 26	GOOF* : Run Error routine
¬ 27	ADDR-OP routine
¬ 28	LINK routine
¬ 29	
¬ 30	Last location of user memory + 1
¬ 31	Contains base of compiled code
¬ 32	Contains end of relocated subroutines
¬ 33	Contains maximum location used by scalars
¬ 34	Base of array stack
¬ 35	Contains compile-time block level of current procedure
¬ 36	Contains run-time block level of current procedure
¬ 37	Base of run-time procedure nesting stack
¬ 38	Run-error recovery cell
¬ 39	Contains segment number
¬ 40	Contains physical right margin for READ
¬ 40+1	Contains physical right margin for PRINT
¬ 40+2	Contains physical right margin for PUNCH
¬ 41	RUN.ERROR switch for end
¬ 42	Run-time error printing mode switch (Chapter 5.RUNERROR)
¬ 43	
¬ 44	
¬ 45	

|variables in ALGOL

|200	READ character pointer for standard buffer
|201	READ right margin for standard buffer
|202	READ left margin for standard buffer
|203	
|204	
|205	PRINT and PUNCH character pointer for standard buffer
|206	PRINT and PUNCH right margin for standard buffer
|207	PRINT and PUNCH left margin for standard buffer
|208	
|209	
|210	Format switch
|211	NAME switch
|212	Page counter
|213	Page header switch
|214	Up-space counter
|215	Left-justify switch
|250	Current READ buffer
|251	Current PRINT buffer
|252	Current PUNCH buffer

System Statements (Chapter 4)

Print Control for Compilation Listing

Statement	Description
PAGE	Eject printer paper to top of next page.
LINE n	Skip n lines.
* SINGLE	Single space the listing.
* DOUBLE	Double space the listing.
* INDENT n or INDENT ± n	Set the left margin of the listing to n or K±n.
$ PRINT <parameter string>	Print or don't print selected parts of the listing.

Miscellaneous

Statement	Description
RIGHT MARGIN n	Scan cards to column n for text.
LIBRARY <identifier>	Fetch <identifier> from the symbolic library.
† n ABCONS	Reserve space for n abcons and n abcons.
SEGMENT n_1, n_2	Treat this program as segment n_1 of length n_2.
RELEASE WHAT	WHAT will no longer be used. Free the space foı compilation.
RELEASE SYMBOLIC LIBRARY	The symbolic ıibrary will no longer be used. Free the space for compilation.
DEBUG n	If n > 0 print the results produced by the three phases of the translator. If n=0 do not print the results.

* not printed

$ may not contain comments

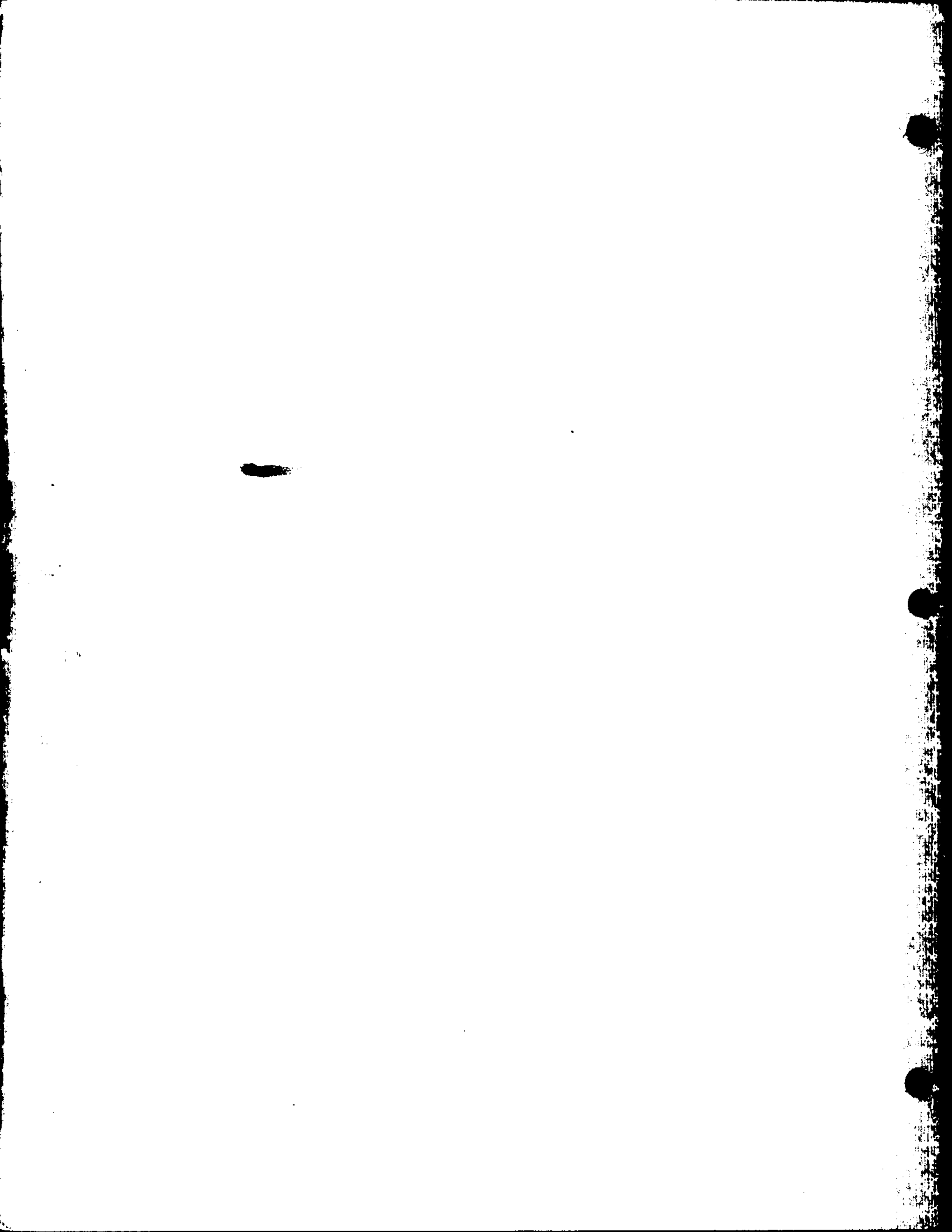
† must occur before the first <u>begin</u>

THIS PUBLICATION IS AVAILABLE AT:

THE BOOK STORE
BAKER HALL
CARNEGIE INSTITUTE OF TECHNOLOGY
PITTSBURGH, PENNSYLVANIA 15213

THE SALE PRICE ($1.25)* REFLECTS THE COST
OF PRINTING AND DISTRIBUTION ONLY.

*(Add $.06 sales tax for purchase in Pa.)
*(Add $.20 postage for mail orders)



CHAPTER I

INTRODUCTION TO ALGOL

ALGOL is an international algorithmic programming language designed for problems whose solution can be expressed in algebraic notation. It is international in that its specifications have been agreed upon by an international committee and it has received wide acceptance throughout the world. It is algorithmic in that it is designed for the natural representation of algorithms. It permits the programmer to write his code in such a way that it is highly readable with an obvious flow of control. The existence of an ever-growing body of published algorithms increases the utility of the language to the user.

The ALGOL language as it exists at Carnegie Tech contains essentially all the features which have been specified for the international language, the only major exception being recursive procedures. Thus the user of ALGOL at Carnegie Tech is using a language with worldwide acceptance and understanding. Our local version has been augmented by certain features not now available in the international language, the most notable of which is an extensive input/output facility.

It is not the purpose of this document to provide the user who is unfamiliar with ALGOL with an introduction to that language, since the literature now includes several very fine works which perform this function admirably. For the user who is learning programming at the same time that he is learning ALGOL, McCracken's Guide to ALGOL Programming is an easy introduction. (See the bibliography at the end of this chapter for a complete citation.) It is used as the text for the introductory programming course at Carnegie Tech. Chapter II of the present work contains a detailed listing of the ways in which the ALGOL system at Carnegie Tech differs from ALGOL as described by McCracken, including references to section numbers in McCracken's text. Unfortunately, McCracken does not give an adequate discussion of ALGOL procedures or of the structure and syntax of conditional statements and expressions.

Bottenbruch's tutorial article in the ACM Journal is a complete introduction, and features good discussions of ALGOL procedures with explanations and examples and of conditional statements and expressions.

For the more experienced programmer who wishes to learn ALGOL, Christian Anderson's text will be worthwhile. Anderson provides a readable introduction to what is important in ALGOL-60.

Another introduction which might be considered is that of E. W. Dijkstra. This is a very complete book describing all of ALGOL. It contains some commentary which is not elsewhere available on the effects of the limited range of number representation in computers. It also contains a good discussion of ALGOL esoterica including Sneaky Procedures.

The basic document which defines the ALGOL language is the Revised Report on the Algorithmic Language ALGOL-60 edited by Peter Naur. This report defines the language completely and unambiguously. It is, however, not easy reading and it is not recommended to the beginner in ALGOL. It is reproduced as Chapter 7 of this report.

Bibliography

Anderson, C., An Introduction to ALGOL 60, Addison Wesley Publishing Co, Inc., Reading, Mass.

Bottenbruch, H., Structure and Use of ALGOL 60, Journal of the ACM, 9, No. 2 (1962), 161-221.

Dijkstra, E. W., A Primer of ALGOL 60 Programming, Academic Press, London, England.

McCracken, D. D., A Guide to ALGOL Programming, John Wiley and Sons, Inc., New York.

Naur, P., editor. Revised Report on the Algorithmic Language ALGOL 60, Communications of the ACM, 6, No. 1 (1963), 1-17.

CHAPTER 2

Notes on ALGOL at Carnegie Tech

INTRODUCTION

ALGOL-60 has been designed to be both a universal language for describing and publishing numerical algorithms, and a programming language for executing algorithms on computing machines. The "reference language" ALGOL-60 has been precisely and elegantly defined in the Revised ALGOL-60 Report (Communications of the ACM, 6, 1 (Jan. 1963)). When ALGOL is actually implemented on a particular computer, however, some changes of notation and some restrictions are usually added to this definition.

The ALGOL translator which has been written at Carnegie Tech for the CDC G-20 computer accepts a language which we call ALGOL-20 to distinguish it from ALGOL-60 when we need to be purists. As a matter of fact, most of the differences between ALGOL-20 and the reference language ALGOL-60 are minor; however, a knowledge of them is needed to use the CIT ALGOL system successfully. In this document, a reference to simply "ALGOL" will always mean ALGOL-60, the reference language.

This chapter describes those aspects of ALGOL-20 which differ from ALGOL-60. As such, it is the primary documentation of our ALGOL system. It is keyed to both the Revised ALGOL-60 report and to the text, A Guide to Algol, by D. D. McCracken. References to the former are by section numbers given in square brackets, and to the latter by section numbers given in round brackets. Thus the paragraph at the top of the next page relates to section 2.3 of the Revised Algol Report and to section 1.4 of McCracken.

SYMBOLS (1.4) [2.3]

The G-20 accepts all of the special symbols of ALGOL-60 except for those shown in the following table:

ALGOL-60		ALGOL-20
⊃	("implies")	Not available, but "¬" may be used with "∨" to obtain the same effect. See page 2.8.
≡	("is equivalent")	Use "=". See page 2.8.
×	(multiplication)	Use "*".
÷		Not available, but "↓" may be used with "/" with the same effect. See page 2.7.
≦		Use "¬>".
≧		Use "¬<".
‘ ’	(string quotes)	Both represented by '. See page 2.3.

In four cases, ALGOL-20 uses a pair of adjacent symbols to stand for a single symbol of ALGOL-60. For example, the ALGOL-60 assignment operator ":=" is strictly a single symbol, but it must be punched into an ALGOL-20 program card as a colon and an equal sign in adjacent columns. There must be no blanks separating the symbols of the pair, and they must both be on the same card. Note: punching them into the <u>same</u> column will give a "hash" of holes which the G-20 will interpret as some other (erroneous) character. The four double-symbol characters are

ALGOL-60 character	ALGOL-20 character pair
:=	:=
≦	¬>
≧	¬<
≠	¬= ("≠" is also allowed)
non-existant	8R
non-existant	8L
non-existant	8F

NUMBERS (2.1) [2.5]

(a) A number, N, in an ALGOL-20 program must either be zero (which may be punched with or without a decimal point) or else its absolute value N must satisfy:

$$1.275_{10}-57 \leqq N \leqq 3.450_{10}+69$$

(b) Because of the nature of the G-20 computer, the distinction between real and integer numbers is unimportant. The programmer may write an integer-valued constant with or without a decimal point (e.g., "34", "34.", or "34.0") without changing the type of arithmetic performed with the constant.

Numbers are represented in the G-20 in "floating point" form with a maximum of 42 binary digits of mantissa, corresponding to approximately 12 decimal digits of precision. If more than 12 digits are written, the extra (least significant) digits will be ignored. (The number is rounded at the 14th octal digit.)

(c) In ALGOL-20, the last character of a real number may be a decimal point; thus, the number "6." is legal.

(d) Octal numbers may be written in ALGOL-20. See Chapter 6e.

STRINGS [2.6]

(a) A string cannot contain a string since ALGOL-20 has no way of distinguishing between the left and right string quotes.

(b) Strings of four characters or less may be used as logic constants and assigned to logic variables. If more than four letters appear in such a string, only the leftmost four are used. Strings of less than four characters are stored right-justified.

IDENTIFIERS AND VARIABLES (2.2) [2.3, 3.1]

(a) Only upper case (capital) letters are available in ALGOL-20.

(b) In ALGOL-20, certain identifiers have special meanings and are therefore reserved. The programmer may never use these reserved ALGOL identifiers as variables or, indeed, for any purpose other than their reserved meanings. These reserved identifiers must be separated from adjacent identifiers

by at least one blank. For example, if the blank between the reserved identifier IF and the identifier "X" were omitted in "IF X > 0", then the ALGOL translator would interpret "IFX" as a single variable identifier; as a result, the statement would have no meaning at all.

The reserved identifiers in ALGOL-20 are

ABS	GO TO	PRINT
ARCTAN	HALF	PROCEDURE
ARRAY	IF	PUNCH
BEGIN	INDEX	READ
BOOLEAN	INPUT	REAL
COMMENT	INTEGER	SIGN
COS	LABEL	SIN
DO	LIBRARY	SQRT
ELSE	LN	STEP
END	LOGIC	STRING
ENTIER	MAX	SWITCH
EXP	MIN	THEN
FALSE	MOD	TRUE
FOR	MONITOR	UNTIL
FORWARD	NAME	VALUE
GO	OUTPUT	WHILE
GOTO	OWN	

Some of these reserved identifiers have no ALGOL-60 equivalent; in particular:

HALF, INDEX, LOGIC (see page 2.5 below)

MAX, MIN, MOD (see page 2.9 below)

NAME, INPUT, OUTPUT, PRINT, PUNCH, READ
(see Chapter 3 - Input/Output)

LIBRARY (see Chapter 5)

FORWARD

MONITOR

FORWARD and MONITOR have not yet been implemented, but will be described when they are available.

All of the ALGOL-60 standard functions are available in ALGOL-20, and their names are reserved identifiers:

ABS	ENTIER	SIGN	(2.4)
ARCTAN	EXP	SIN	[3.1.4]
COS	LN	SQRT	

See Chapter 5 for further information on these functions.

"TO" is reserved only when it follows immediately after the reserved identifier GO. In any other context, "TO" may be used as an ordinary

identifier by the programmer. See page 2.10 of these notes.

In addition to the reserved words listed above, ALGOL-20 includes a set of "privileged" identifiers which have built-in meanings without being declared; they are, in effect, declared by the translator in a block head outside of the outer-most block of the program. Therefore, if the programmer does not wish to use one of these identifiers in its privileged meaning, he may simply ignore the fact that it is privileged and declare and use it as he would any non-special identifier. Further, if a privileged identifier is declared in an inner block, it resumes its privileged meaning as soon as the end of the inner block is passed. These identifiers are listed and their meanings are explained in Chapter 6d. Identifiers may be added to this list by the Computation Center at some future time. Since they are not reserved, additional privileged identifiers cannot accidentally interfere with identifiers written into a current ALGOL program.

(c) Spaces may not appear within an identifier in ALGOL-20. The programmer may, however, freely sprinkle periods (.) within identifiers to separate them into words and improve the readability of the program. These periods are ignored by the ALGOL-20 translator; therefore, the following are all instances of the same identifier::

READACARD

READ.A.CARD

R.E.A.D.A.CARD..

(d) ALGOL-20 allows both simple and subscripted variables of type half, and logic, as well as real, integer, and Boolean. Also, simple variables may be of type index.

Real variables are stored in the G-20 with a precision of 42 binary digits, requiring two successive memory cells per variable. Half variables are stored with a precision of only 21 binary digits (about 6 significant decimal digits) and occupy only a single location, but otherwise act as real variables. Therefore, the programmer may use half variables to gain memory space at the expense of precision.

Logic variables are unsigned 32 bit G-20 logic words, which may be used for bit and character manipulation processes. They may be used in either arithmetic or Boolean expressions. Simple variables of type index will be assigned to G-20 index registers but act otherwise as variables of type

integer. The uses of logic and index variables are complex to explain but obvious to those ALGOL programmers who are also knowledgeable in G-20 machine language For more information see Chapter 6e.

(c) The value of a real or half variable must either be zero or else lie within the range given below:

$$\text{real:}\quad 1.275_{10}-57 \leqq \text{abs(R)} \leqq 3.450_{10}+69$$
$$\text{half:}\quad 1.275_{10}-57 \leqq \text{abs(H)} \leqq 1.645_{10}+63$$

integer and index variables will always take on integer values in the range

$$-2097152 < I < 2097152 \quad (\doteq 2^{21})$$

logic variables are always positive. If used as strings, they are four or less characters in length, and if used as numeric quantities they are restricted to

$$0 \leqq L < 4294\,967296 \quad (\doteq 2^{32})$$

The values of Boolean variables must be either true or false.

The G-20 replaces by zero any non-zero arithmetic result which is smaller than $1.175_{10}-57$ in magnitude; this situation is called an underflow. An intermediate arithmetic result which is greater than $3.450_{10}+69$, the largest number representable in the G-20, is called an overflow. An overflow during execution of the object program will cause the run-time error message "RUN ERROR-EXPO" to be printed, and terminate execution of the program (unless error recovery is in use). See Chapter 6b for further details on run-time errors.

An exponent overflow cannot occur during translation of the ALGOL source program; violation of the restrictions on ALGOL-20 numbers given above will cause a normal syntactic error message which will not, however, terminate translation.

The number $3.450_{10}+69$ is the upper limit for the result of each individual arithmetic operation in the evaluation of any arithmetic expression, regardless of the types of the variables in the expression. However, if the result of the expression is assigned to a half variable, then a value greater than $1.645_{10}+63$ will result in an exponent overflow message as explained above. A value assigned to an integer variable, on the other hand, will be truncated modulo $2^{21} \doteq 2097152$; while a value assigned to a logic variable will be truncated modulo 2^{32} (and given a positive sign); in either case, no overflow message will occur.

ARITHMETIC EXPRESSIONS (2.3) [3.3]

(a) In ALGOL-20, the asterisk ("*") is used in place of the multiplication sign ("×") of ALGOL-60.

(b) ALGOL-20 arithmetic expressions may contain the truncation operator "↓" defined mathematically by

$$\downarrow X = \text{sign}(X) * \text{entier}(\text{abs}(X))$$

That is, ↓X is simply the integer part of X if $X \geqq 0$, and is -(integer part (-X)) for $X < 0$. Thus, ↓(1.7) = 1, ↓(-1.7) = -1. Truncation is performed modulo $2^{32} = 4294967296$; for example, ↓4294967298 = 2.

The truncation operator is unary, having exactly one operand which is the complete expression immediately to the right of the "↓" symbol. The precedence of "↓" is very high, so that "↓" will be executed before "↑" or any other arithmetic operation (unless parentheses are used to force a difference order). For example, "↓X/Y" means (↓X)/Y and "X↑↓Y" means X↑(↓Y). (Truncation is done by an add-logical in mode zero of zero.)

(c) The truncation operator, "↓", can be used to get the effect of the integer divide operation, "÷", which is not available in ALGOL-20. If I and J are integer variables, then

$$I \div J = \downarrow(I/J)$$

Notice that the "↓" operator can operate on any integer or real expression, and is therefore more general than "÷".

(d) When a variable of type half appears in an arithmetic expression, the rules for determining the type of the result are exactly as if the half variable had been of type real. In fact, full precision (42 bit) floating point arithmetic is always performed on all variables other than Boolean and logic in the G-20.

(e) The "+" and "-" can be used either as binary operators or else as unary operators. When "+" and "-" are used as unary operators with "↑" in the combination "↑+" or "↑-", parentheses around the exponent may be omitted.

The following table shows some examples of this rule:

The ALGOL-20 Expressions	Means: (both in ALGOL-20 and ALGOL-60)
X↑+Y	X↑(+Y)
X↑-Y	X↑(-Y)
X↑↓-Y	X↑(↓(-Y))

(f) The precedence of operators and relations in ALGOL-20 is

↓ (done first)

mod

↑

- + (used as unary operators)

/ *

- + (used as binary operators)

≠ = ¬> ¬< > <

¬

∧

∨ (done last)

That is, unless parentheses force a different order, ↓ will be performed, then mod, then ↑, and so on. The unary operators + and - are special cases. Unary + is ignored. Unary - is performed on the expression on its right whose operators have higher precedence than it. For example,

q mod - a ↑ b mod c * d

is

q mod((-(a ↑ (b mod c))) * d)

BOOLEAN EXPRESSIONS (3.6) [3.4]

(a) The Boolean operator "⊃" ("Implies") is not available in ALGOL-20. However, for any Boolean expressions B_1 and B_2, the ALGOL-60 expression $B_1 \supset B_2$ may be replaced by either of the equivalent forms:

$$\neg B_1 \vee B_2$$

$$\neg (B_1 \wedge \neg B_2)$$

(b) ALGOL-20 substitutes the equality symbol "=" for the Boolean equivalence operator "≡". Note that the ALGOL 60 report gives ≡ very

low precedence. ALGOL-20 cannot distinguish between ≡ and = and thus gives them the same precedence. Thus $A \wedge B = C \vee D$ is taken as $A \wedge (B \equiv C) \vee D$, and parentheses must be used if any other meaning is intended.

STANDARD FUNCTIONS (2.4) [3.2.4]

ALGOL-20 has three built-in operators, MAX, MIN and MOD, which are not in ALGOL. These are defined mathematically as follows, where $E_1, E_2, \ldots E_n$ are arithmetic expressions.

MAX (E_1, E_2, ... E_N) = the largest algebraic value of the N expressions;

MIN (E_1, E_2, ... E_N) = the smallest algebraic value of the N expressions;

E_1 MOD E_2 = $E_1 - E_2 * \downarrow (E_1/E_2)$

MAX and MIN may have any number of expressions as arguments. Note that MOD is written as an operator between its two arguments. The above definition for MOD holds for all values of E_1 and E_2, but in the case where both arguments are positive integer-valued expressions, then E_1 MOD E_2 is the remainder for E_1 divided by E_2 (and $\downarrow(E_1/E_2)$ is the integer quotient). Although E_1 and E_2 each appear twice in the definition, they are actually evaluated only once.

ASSIGNMENT STATEMENTS (2.5) [4.2]

(a) In addition to the ":=" operator of the reference language, ALGOL-20 allows the left arrow ("←") as an assignment operator. The left arrow has the same meaning as ":=", except when a non-integer expression is assigned to an <u>integer</u> variable. The assignment statement

<integer variable> ← <non-integer expression>

result in truncating the value of the expression to an integer <u>without rounding</u>. If ":=" is used instead, the value will be rounded to an integer in conformity with the reference language; however, the "←" operator produces more efficient object code.

(b) In a multiple assignment statement, the "left part" variables need not all be of the same type. For example, the sequence

```
REAL X ; INTEGER I, J ;
I ← X ← J := 3.7*X;
```

is allowed in ALGOL-20. The rule given in (a) above determines for each integer left part variable whether or not rounding will occur.

LABELS AND GO TO STATEMENTS (3.2) [4.3]

(a) Only identifiers may be used as labels in ALGOL-20; integer labels are not permitted.

(b) In ALGOL-20, GOTO and GO are both reserved identifiers, and TO is ignored when it follows after GO. Hence

```
GO TO Label
GOTO Label
GO Label
```

are all equivalent and permissible.

CONDITIONAL STATEMENTS (3.3) [4.5]

(a) Because of character set restrictions, ALGOL-20 must make the following substitutions for relational operators:

ALGOL-60	ALGOL-20
≧	¬<
≦	¬>

In addition, both "≠" and "¬=" are allowed in ALGOL-20.

(b) There are some complex syntactic construction which were allowed by the original ALGOL-60 report but were subsequently found to be ambiguous or controversial. One such ambiguity arises when a *for* statement comes within the scope of an *if* clause.

(1) Consider the following construction:

```
if ... then
  for . . do
    begin
      if ... then <unconditional statement>
              else <statement>
    end;
```

If the "begin ... end" pair is omitted, this construction becomes ambiguous since the phrase "else <statement>" could belong to either the inner or the outer if clause. ALGOL-20, in agreement with the 1962 revision of ALGOL-60, allows the "begin ... end" pair to be omitted, and considers "else <statement>" to belong with the second <if clause>; i.e., the construction is treated as if the "begin ... end" pair were actually present.

(2) The following construction:

```
if ... then
  for ... do <unconditional statement>
    else <statement>
```

is not actually ambiguous. However, the revision of ALGOL-60 syntax which took care of case (1) also had the undesirable effect of outlawing construction (2) which is perfectly respectable. Therefore, ALGOL-20 will allow (2) but will print a "Note 7" (see Chapter 6b) to point out that it is inconsistent with revised ALGOL-60 syntax.

CONDITIONAL EXPRESSIONS (3.5)

(a) ALGOL-20 allows certain constructions with conditional expressions which are unambiguous but illegal in revised ALGOL-60. The ALGOL-20 translator will flag any of these constructions with a "Note 4" message (see Chapter 6b) to call the programmer's attention to the violation of ALGOL syntax.

In ALGOL-20 the right-hand operand of a binary operator may be a conditional expression without parentheses; e.g., the second set of parentheses may be omitted in:

(if X > 0 then X else Y) + (if Y > 0 then 3 else X)

Note, however, that omission of the first set of parentheses, surrounding the conditional expression which is the left-hand operand of the binary operator "+", would change the meaning to the following:

if X > 0 then X else (Y + if Y > 0 then 3 else X).

Similarly, the following construction is legal in ALGOL-20:

X * if A > B then 3 else Y + Z

but will cause a "Note 4". It will be interpreted as:

X * (if A > B then 3 else (Y + Z)) .

ALGOL-20 allows the analogous constructions with binary Boolean operators and conditional Boolean expressions, and with relational operators and conditional arithmetic expressions. An example of the last is the Boolean expression

(if BOOL then X else Y) < if BOOL then 3 else Z

The expression with the first set of parentheses omitted would be interpreted as

if BOOL then X else (Y < (if BOOL then 3 else Z))

FOR STATEMENTS (4.1) [4.6]

(a) A left arrow may be used instead of ":=" in an ALGOL-20 for clause; "←" will truncate and ":=" will round each implicit assignment to a for variable of type integer.

(b) The value of the controlled variable is not undefined upon normal exit from an ALGOL-20 for statement. The value of the for variable upon exit depends upon the form of the last element in the for list, and is in general just what would be obtained if the equivalent basic programs (see section 4.1 of McCracken or section 4.6.4 of the report) were substituted for the for statement. Thus, upon exit from an until or while form of for list element, the for variable has the first value for which the final test failed. For example:

FOR I ← 1 STEP 1 UNTIL 10 DO S ;

leaves I = 11 when the for list is exhausted and control passes to the next statement.

(c) A fourth form of for list element is permitted in ALGOL-20:

FOR V ← E_1 STEP E_2 WHILE B DO S;

where E_1 and E_2 are arithmetic expressions, B is a Boolean expression, and

S is any statement. This is equivalent to the simple program:

```
        V  ←  E1 ;
LOOP:   IF  B  THEN
          BEGIN
             S   ;
            V  ←  V + E2;  GO TO LOOP
          END ;
```

Notice that if the Boolean expression B is: $(V - E_3) * (E_2) \leqq 0$ then the new step ... while form of for list element is identical to the step ... until form. However, when (as is usual) the sign of the step expression E_2 is known to the programmer, the step ... while form (omitting the multiplication by E_2) will be more efficient in both space and time.

ARRAYS (5.2, 5.3) [5.2, 3.1.4]

(a) ALGOL-20 arrays may be of type integer, real, Boolean, half, or logic. Index arrays are not permitted.

(b) A non-integer value of a subscript expression in ALGOL-20 is not rounded, only truncated. This may lead to hard-to-detect errors. For example, suppose that the result computed for a subscript expression is 3.9999... instead of 4 because of round-off error; this value will be truncated to 3, referring to the wrong element of the array. Thus, the plausible program:

```
FOR  X  ←  0  STEP  0.2  UNTIL  1.0  DO
     A  [5 * X]  ← X ;
```

may not work correctly because of the round-off error in 0.2 which cannot be exactly represented in a binary computer like the G-20. The following alternative will work:

```
FOR  I  ←  0  STEP  1  UNTIL  5  DO
     A [I]  ←  I/5 ;
```

(c) The speed of execution of an ALGOL-20 program does not depend upon the lower or upper bounds of an array subscript, upon the order of the dimensions, or upon the types of variables appearing in subscript

expressions, however, the number of memory cells required by an array does depend upon the order of the dimensions; the least number of cells is required if the longer dimension is listed last.

OWN VARIABLES (6.6) [5.0]

Own arrays may be used in ALGOL-20, but they must have fixed subscript bounds so that storage may be allocated to them before execution begins; that is, "dynamic own arrays" are not allowed.

Own simple variables and own arrays are initialized to zero (or false, in the case of Boolean quantities or ' ' in the case of logic quantities) before execution begins.

PROCEDURES

(a) Parameters (7.4) [4.7]

When the first occurrence of a label in a block is as an actual parameter in a procedure call, then the ALGOL-20 processor must be forewarned that this identifier is a label. This requires that the label identifier appear in a label declaration in the block head. For example:

```
BEGIN
   INTEGER I, J; LABEL L;
   PROC (X, L) ;
L : I ← I + 1 ;
END ;
```

This is the only circumstance in which a label declaration is required in ALGOL-20.

(b) Specifications (7.5) [5.4.5]

All formal parameters in an ALGOL-20 procedure declaration must appear in the specification part of the procedure heading.

(c) Recursive Procedures (7.7)

Recursive procedures are not now available in ALGOL-20.

(d) Arrays, switches, and labels cannot be called by value.

CHAPTER 3

Input/Output Statements

3a. Introduction

The official ALGOL-60 language does not include input/output statements. Thus, ALGOL-60 can be used to describe computational algorithms but not the process of reading input data from punched cards, magnetic tape or disc, or the process of outputting intermediate and final answers onto printed pages, punched cards, magnetic tape or disc. Each ALGOL translator, therefore, must contain its own scheme for programming input and output operations.

ALGOL-20 includes an input/output ("I/O") system derived from the system used previously in the GATE language at Carnegie Tech.[1] The following pages contain both an introductory explanation and a complete technical description of ALGOL-20 statements for reading data cards and for printing and punching answers.

Chapter 3b is a primer on ALGOL-20 I/O which takes a particular example of printed output and builds up its solution. It is introductory in nature, and concerns only printing. Punching requires only simple extensions of the concepts used in printing. Chapter 3c is a primer on READ which includes a completely worked-out example. Chapter 3d contains a complete summary of all input/output instructions.

ALGOL-20 also contains provision for reading and updating files of information stored on magnetic tape or disc. This mechanism is related to the card reading, printing and card punching statements, but involves additional complexity. It is described separately in Chapter 6g.

---

1 The GATE input/output system is described in the manual: "20-GATE: Algebraic Compiler for the Bendix G-20", Carnegie Tech Computation Center, September 1962. The general principles of the ALGOL-20 input/output system were the subject of a paper presented by A. J. Perlis at the Working Conference on Mechanical Language Structures, August, 1963, published in _Comm. A.C.M._, 7 (Feb. 1964) p. 89.

CHAPTER 3b

Primer on ALGOL-20 Input/Output

Consider the task of programming a computer to print answers. To control printing, such a program must specify two distinct kinds of information:

(1) Which values are to be printed, and

(2) The format in which the values are to appear on the page.

To supply these two kinds of information, ALGOL-20 contains two types of statements: NAME statements, which select the values to be printed, and PRINT statements, which specify the printed format for these values. "NAME" and "PRINT" are reserved identifiers in ALGOL-20. In general, each NAME statement is paired with a PRINT statement and the two are used in parallel to control printing; each value specified by the NAME statement must be matched with a format specification from the PRINT statement.

The remainder of Chapter 3b is divided into sections, as follows:

A. The NAME Statement: Introduction

B. The Format Program: Introduction

C. The Print Buffer

D. An Example of Print Format

E. Replicators: Introduction

A. The NAME Statement: Introduction

A NAME statement in ALGOL-20 has the following form: The reserved identifier NAME followed by a pair of parentheses enclosing a name list. For printing (or punching), the name list is a list of values to be output and therefore is simply a list of arithmetic expressions (separated by commas):

```
NAME ( < Arith Expr >,..., < Arith Expr >)
```

When a value is needed by a PRINT statement, the value of the next expression in the NAME list is computed and supplied to the appropriate PRINT instruction. Expressions in the NAME list are evaluated in left to right order, and the corresponding values are printed in the formats specified by the PRINT instructions.

For example, to print the values of the ALGOL variables A, B and C and also the value of the expression $\sqrt{B^2 - 4AC}$, the programmer may use the NAME statement:

```
NAME  (A,B,C,SQRT(B ↑ 2 - 4*A*C) )
```

along with an appropriate PRINT statement.

NAME statements may be more complicated. For example, they may contain for clauses and other forms of replicators which repeat the selection of values in a manner analogous to the repeated execution of an ALGOL statement by an ALGOL for clause. Replicators are discussed in Section E.

B. The Format Program: Introduction

Suppose that the value 1.7 has been computed and is to be printed by an ALGOL program. This number could be printed in any one of many different formats; for example, one of the following forms might be appropriate in a specific case:

1.7   +1.7   +00001.700 $.170_{10}{+01}$ $1.70_{10}{+00}$ $17000_{10}{-04}$

However, there is more to format control than specification of the forms of individual numbers. Answers are generally to be printed in a readable manner: separated by blank columns and accompanied by suitable headings and titles to identify the printed results. Therefore, a PRINT statement must give the programmer control over the position of each number and title on the line, the assignment of numbers to different lines, the spacing of printed lines on the page, and the sequencing of pages, as well as the form of numbers.

To control all these aspects of format, ALGOL-20 contains a special "format language", which is used within PRINT statements. A series of instructions in this format language forms a format program. The individual instructions within a format program are separated by commas.

The format language uses some of the same characters that ALGOL uses, but with different meanings. Therefore, special brackets must be placed around each format program to set it apart from the ALGOL program in which it is embedded. Unfortunately, there are no unused symbols available in the G-20 alphabet for these format brackets, so we use "<" (less than) and ">" (greater than) for this purpose. The syntax of a PRINT statement is such that "<" and ">" symbols surrounding format programs cannot be confused with the same symbols in Boolean expressions.

The simplest form which a PRINT statement may have is the reserved word PRINT followed by a pair of parentheses which enclose a single format program, or enclose a series of format programs separated by commas. Each format program is itself enclosed in "<" and ">" brackets. The following PRINT statement, for example, contains a single format program which consists of five format instructions:

```
PRINT ( < P, 37C, 'A=', + 2D.3Z, 2E >)
```

The meanings of these instructions will be explained below. The effect of this PRINT statement would not be changed if each format instruction were enclosed in format brackets, so that the PRINT statement contained five format programs each consisting of a single format instruction:

```
PRINT (< P >, < 37C >, < 'A=' >, < +2D.3Z >, < 2E >)
```

C. The Print Buffer

Associated with the G-20 printer is a block of 120 consecutive cells in memory, called the print buffer. These cells, numbered 1, 2, 3,..., 120, correspond to the 120 physical print positions or "columns" in a line of printing.

The process of printing takes place in two steps: First, a format program in a PRINT statement places the characters to be printed into the print buffer, each character being placed into the cell corresponding to the column in which it is to be printed. In this manner, the format program builds up an "image" of the line to be printed. Second, when the entire line has been formed, a format control instruction must be executed to send all 120 characters from the print buffer to the printer and actually print the line on the paper. The format instruction which is generally used for the latter purpose is 'E', which is mnemonic for Execute. The E instruction prints the image in the print buffer and afterwards automatically "erases" the print buffer (i.e., clears it to 120 blank characters) in preparation for the next line.

The print buffer behaves like other memory cells: Storing a new character into a buffer cell replaces the character which was there previously, while sending a character to the printer to be printed does not (necessarily) erase it from the print buffer. In particular, the control instruction 'W' executes the same printing operation as 'E' but does not erase the buffer afterwards. Thus, the programmer may, if he wishes, save part (or all) of the print image for printing on successive lines.

Associated with the buffer is a pointer called the "character pointer" or "CP". The value of CP is always the number of the print buffer column into which the next character will be stored by a format instruction. As each character is stored, CP is automatically stepped ahead (to the right) by one so that successive characters are stored in left-to-right order into successive cells. Therefore, execution of a format instruction which stores characters into the print buffer automatically leaves CP set to the first column after the last character stored. For example, if CP is

47 and a format instruction stores a number requiring 5 columns, CP will be left at column 52.

Another pointer contains the "left margin" or "LM". The value of LM is the number of the left-most column into which characters may be stored. Execution of the instruction "E" leaves CP reset to the value of LM. (Execution of "W" leaves CP unchanged.) There is also a pointer which contains the "right margin" or "RM" -- the number of the right-most column into which characters may be stored. Initially, LM and RM have the values 1 and 120 respectively. Before each character is stored into the print buffer, a check is made to insure that:

$$LM \leq CP \leq RM$$

If this relation does not hold, an "E" is automatically executed: the characters already in the buffer are printed, the buffer is cleared, and CP is reset to the value of LM. The character is then stored into the buffer. The mechanism for changing LM or RM is explained in Section E of Chapter 3d.

D. An Example of Print Format

A particular print program will now be discussed in detail. Assume that an ALGOL program computes all the values in a 40 x 10 array (40 rows x 10 columns) COEF; these 400 values are to be printed along with a value of a simple variable DELTA. A sample of the desired printing is shown on page AL.3b.7.

The printing begins with a title, "ADJUSTED COEFFICIENT MATRIX", which starts in print position 37 of the first line on the page. The "1" in the next printed line is in column 17, the "2" in column 28, etc. The row numbers, down the left-hand column, are in print positions 6 and 7. Each matrix element occupies nine positions in the printed line and is separated from its neighbors by two blank spaces. The numbers to be printed are all less than 1000 in magnitude, and four digits are to be printed to the right of the decimal point. A minus sign is to be printed

immediately before the first digit if the number is negative. The value of DELTA is to be printed with two significant digits in "scientific notation", with a power of ten, as shown. No sign is to be printed for DELTA. The step by step construction of the necessary NAME and PRINT statements for printing this example follows.

First, consider printing the title. Three different types of formating operations are needed for this purpose:

(1) An instruction is needed to begin printing at the top of a page.

(2) An instruction is needed to indicate that the information is to be printed starting in column 37.

(3) Instructions are needed to specify the information to be printed.

Since the title is a fixed string of alphabetic information, it is convenient to include it entirely in the PRINT statement, with no corresponding value in a NAME statement. In fact, if only fixed information such as a title were to be printed, no NAME statement would be needed with the PRINT statement; this is an important exception to the general rule that NAME and PRINT statements come in pairs.

To specify a title or any other fixed string of alphabetic characters to be printed, we use a format instruction called an alphanumeric string instruction. This is simply the string of characters to be printed, enclosed in quote marks. Such an instruction can thus be used to print any character except the quote mark, since a quote within the string cannot be distinguished from the quote terminating the string. (A special format instruction is provided for printing a quote mark -- see page AL.3d.6) The alphanumeric string instruction used to specify the title is:

'ADJUSTED COEFFICIENT MATRIX␣-␣␣DELTA␣=␣'

(Here and in the sequel we use the symbol "␣" to represent a blank column, where it is necessary to emphasize that a column is to be blank.) Blank is a legitimate alphabetic character, so all blanks appearing in the alphanumeric string instruction will appear as blank columns in the title as printed.

ADJUSTED COEFFICIENT MATRIX - DELTA = 5.0 *-04

	1	2	3	4	5	6	7	8	9	10
1	1.7902	-0.0000	0.0000	0.0008	-0.0000	-0.0000	0.0034	-0.0573	-11.1610	323.8654
2	-0.0001	93.9153	0.0005	7.2796	0.0007	-16.7266	1.7892	-0.0202	-2.6406	0.0003
3	-0.6511	-0.0077	-0.0137	-0.0006	-0.0055	-79.3906	-98.8134	-4.5811	-0.0446	-0.7871
4	-1.6910	0.5956	-0.0025	0.0005	-20.9818	-0.1082	0.1210	1.8640	-0.0102	0.0004
5	-0.1198	-98.2422	-0.0000	0.0001	975.0974	0.0002	-0.0228	-45.1398	-0.0472	-17.3674
6	5.4653	0.5186	1.7492	0.0041	-0.0001	-19.0514	-0.0001	-0.0061	-0.0377	0.0591
7	-5.2393	-21.0519	-0.0180	72.8323	-0.0000	-0.1028	-0.0047	-29.7404	-196.5030	7.9580
8	0.3571	-0.0060	0.0000	-1.9352	777.2626	0.0002	-0.0692	52.0718	-0.0826	0.0000
9	45.4917	-9.7117	0.1306	-0.2042	-0.0002	0.0014	0.1669	0.0013	0.0008	0.0003
10	0.1561	0.1567	0.0002	3.6563	-0.2210	-18.2905	-22.9516	146.4738	0.0001	0.0000
11	0.0012	-0.0022	-143.7867	179.7959	0.0672	-0.0025	368.0110	-594.3002	-1.7935	-7.2878
12	-0.0345	-0.0285	-16.0073	0.0035	-0.0000	-0.0000	-0.0001	0.0003	0.2284	1.7218
13	-2.3244	-0.0100	0.0001	85.3077	0.0000	-0.0041	-0.0000	-0.0094	1.5433	0.0136
14	-0.0004	0.1944	0.2522	-0.0040	-0.0067	110.9960	-3.5136	20.5636	-201.2119	-0.4635
15	-0.1697	-66.2361	0.0017	665.2994	0.0021	0.0000	0.0115	-0.0000	-0.0804	-0.0000
16	0.0004	-481.6797	-0.0232	212.5706	0.0005	0.0954	-2.0125	3.0202	513.5410	0.9989
17	-0.3790	-814.4740	0.1218	462.6379	-19.9396	-46.4429	359.0704	0.0014	-6.7067	0.0001
18	127.0418	-742.4814	-0.0001	0.0004	-925.1035	-3.1928	0.0097	-0.0013	-0.0423	-0.9347
19	-0.0036	427.1130	-0.0659	-0.0003	0.0847	29.2204	-33.6923	-0.0219	-4.2031	-14.3640
20	0.0010	0.2172	0.3396	-3.1782	-0.0000	-0.0226	575.6602	-35.2607	2.2267	0.0000
21	0.4675	0.0039	-92.6274	0.0000	-0.0074	-14.9826	0.5947	0.3146	0.0000	1.2663
22	0.0018	6.5445	0.6704	26.1315	-0.0139	-0.0033	-112.8084	0.0000	263.9816	1.0316
23	320.5485	-0.4436	-212.2199	0.0001	1.1127	0.0805	-219.9632	226.8146	0.0001	-0.0017
24	0.1595	-0.0020	0.0012	-0.0088	77.9889	-0.0002	0.0034	0.3729	-0.0000	0.6368
25	-0.0010	-0.8177	-88.2311	-0.0094	0.0043	-1.0382	-337.7289	0.0079	0.5487	-1.5120
26	0.2388	0.0001	0.0089	-0.0001	535.3546	-0.0089	-13.3044	0.0133	0.0000	0.0467
27	-0.0002	-0.0006	-0.1319	-214.9891	-11.0611	0.0013	75.8273	228.7995	-574.2057	-0.6027
28	0.0378	-26.4622	-0.0000	0.0003	0.0005	0.2778	-0.0994	-0.0268	0.0001	0.0039
29	0.9747	0.5692	-2.4955	198.4666	-907.9086	0.0116	-0.0156	-0.0017	580.9429	-0.0142
30	0.0071	-0.0000	-0.0003	1.1572	1.2592	-484.6675	-0.0000	0.0000	-85.1370	-0.0652
31	0.0035	63.2521	-3.7464	-0.0002	-0.0197	-0.0093	-0.0033	0.0017	-0.0001	0.0004
32	-307.9847	0.0004	-632.2322	139.1301	0.0030	-810.0790	33.7184	-0.0098	0.0005	64.7952
33	0.0009	-0.0983	-496.9972	1.7182	-0.0002	0.1562	138.8615	0.0224	0.0005	0.7802
34	110.3654	0.0000	-0.0079	0.0043	0.0000	-3.9576	-0.0060	0.0002	-0.0002	0.2098
35	0.0003	-488.2567	0.0115	-843.1841	1.4976	0.2209	-0.0420	0.0068	-45.7289	0.0001
36	-0.0204	-0.0001	0.5394	-4.4282	-40.3666	0.0031	0.0000	-136.9766	-0.0018	0.0021
37	0.0002	-0.0000	-165.2740	-457.8515	-0.0001	0.0000	56.5357	0.0142	66.8979	-0.1896
38	21.5920	-0.0054	2.9670	-3.4496	0.0499	-3.6325	-1.2954	0.2277	0.0001	0.0000
39	-0.0000	-0.1920	914.2467	-839.1647	0.0000	-0.0000	378.6081	-0.0000	216.4519	7.7986
40	-26.0042	0.4164	-348.2117	-0.0083	285.7574	-112.9313	-0.1943	-548.3568	8.6184	0.0011

This string is to be stored in the print buffer starting at column 37, so CP must be set to 37 before the alphanumeric string instruction is executed. The format instruction to do this is "37C"; here "C" is mnemonic for "Column". Generally, executing an instruction of the form "nC", where n may be any integer in $1 \leq n \leq 120$, will have the effect of setting CP to column n: $CP \leftarrow n$. The format program <1C, 37R> might also have been used. 1C sets CP to column one, and 37R moves CP 37 columns to the Right. Similarly, nL moves CP n columns to the Left. To summarize:

nC	has the effect	$CP \leftarrow n$
nR	has the effect	$CP \leftarrow CP + n$
nL	has the effect	$CP \leftarrow CP - n$

Therefore, the following format program will set CP to 37, place the 40 characters of the string into print positions 37 to 76 of the print buffer, and then print the buffer:

```
<37C, 'ADJUSTED COEFFICIENT MATRIX  -  DELTA =  ', E>
```

This could just as well have been written as three successive format programs by putting brackets around each instruction:

```
<37C>, <'ADJUSTED COEFFICIENT MATRIX  -  DELTA =  '>, <E>
```

but the first form is easier to punch. The instruction necessary to store the value of DELTA into the print buffer is still missing. For reasons which will be discussed later, the appropriate numeric instruction is 1D.1ZL. Further, the title is to be printed at the top of the page. The format instruction used to upspace the paper to the top of the next page is "P". Thus, a complete ALGOL-20 program to print the first line of the example might be

```
NAME(DELTA);  PRINT(<P, 37C, 'ADJUSTED COEFFICIENT MATRIX  -  DELTA =  ',
1D.1ZL, E>);
```

Equivalently, the following might be used:

```
PRINT(<P, 37C, 'ADJUSTED COEFFICIENT MATRIX  -  DELTA =  '>);
NAME(DELTA);  PRINT(<1D.1ZL, E>);
```

Next consider the format for printing the number DELTA and the numbers of the matrix itself. Numeric instructions are those instructions which place numbers into the print buffer; these numbers are values which are obtained from the evaluation within the parallel NAME statement.

A numeric instruction may be regarded as giving a "picture" of the number to be printed. Generally, the following items must be specified to define a number format:

(1) The form for printing the sign, if at all.

(2) The number of places, if any, to the left of the decimal point, and whether leading zeroes are to be inserted or left as blanks.

(3) The decimal point, if any.

(4) The number of places, if any, to the right of the decimal point, and whether trailing zeroes are to be inserted or left as blanks.

(5) The "exponent part" (power of ten), if any.

Items (2), (3) and (4), defining the format of the numeric part of the number without sign or exponent, are specified by the number form portion of a numeric instruction. Item (1), the sign, is specified by the sign part, which is part of the prefix, while item (5) is specified by the suffix of the numeric instruction. The form of a numeric instruction then is given by:

<numeric instruction> ::= <prefix> <number form> <suffix>

(We will see later that the prefix includes, in addition to the sign part, a part which controls the printing of dollar signs.) The number form gives a simple picture of the basic form of the number; as an illustration, the matrix values in the example may be printed with the number form:

3D.4Z

Here "3D" indicates three Digits to the left of the decimal point, with leading zeroes replaced by blanks; the period is a picture of the decimal point which is to be printed; and "4Z" means four digits to the right of the decimal point, with trailing Zeroes printed. For example, the number 3.74 will be printed:

by 3D.4D in the form ␣␣3.74␣␣

by 3D.4Z in the form ␣␣3.7400

by 3Z.4Z in the form 003.7400

by 3Z.4D in the form 003.74␣␣

by 3Z in the form 004.

by 3D in the form ␣␣4

All blanks stored are shown explicitly by ␣. Notice in the last two examples that the number was rounded by adding five to the first digit not printed, and then truncating the result. The syntax of number form is as follows:

<number form> ::= <integer part> | <integer part>. |
<integer part>.<fractional part> | .<fractional part>
<integer part> ::= <unsigned integer> D | <unsigned integer> Z
<fractional part> ::= <unsigned integer> D | <unsigned integer> Z

If the integer part (fractional part) appears, at least one digit will be printed before (after) the decimal point. For example, the number zero printed with the numeric primary 3D.2D appears as ' 0.0 '. The total number of digits specified must be less than 15.

In our example, DELTA is to be printed with one digit preceding and one digit following the decimal point, so it may be printed with any one of the following number forms:

1D.1D 1D.1Z 1Z.1D 1Z.1Z

The program which actually printed the sample included 1D.1Z to print DELTA.

The prefix includes the sign part to specify the form for printing the sign of the number. If no sign is to be printed, this part is left empty, as is the case for DELTA. The array elements are to be printed with a minus sign immediately preceding the first significant digit of each negative number. The sign part to use in this case is "-". If in addition plus signs were to be printed before each non-negative number, the prefix "+" would be used instead.

The suffix portion of a numeric instruction is used to supply supplementary information, such as scaling the number, printing an exponent or special spacing. The format for the array elements is completely specified by the prefix and the numeric primary portions, so the proper numeric

instruction is -3D.4Z. DELTA is to be printed in scientific notation: shifted so that the left-most digit is non-zero (if possible) and the resultant exponent printed. The suffix "L" provides such printing, so the numeric instruction 1D.1ZL is to be used to print DELTA.

E. Replicators: Introduction

In principle, everything which is necessary to print the example has now been discussed. However, writing or punching the NAME and PRINT statements for the example using only the NAME and PRINT machinery discussed so far would be very lengthy and tedious. For example, it seems as if the NAME statement would have to be a simple list of all of the 401 variable names DELTA, COEF[1,1], ..., COEF[40,10], while the PRINT statement would have to contain 401 distinct numeric instructions in addition to alphanumeric string instructions and control instructions. What is needed is a "loop" mechanism analogous to the ALGOL for statement; this mechanism is provided by <u>replicators</u>.

An ALGOL program which would operate in some way upon each element of each row of the matrix COEF would presumably have the form of two nested <u>for</u> statements:

```
FOR I ← STEP 1 UNTIL 40 DO
   FOR J ← 1 STEP 1 UNTIL 10 DO
      something with COEF[I,J] ;
```

This is essentially the form which is used in the NAME statement; the "action" to be performed on COEF[I,J] is simply "naming" its value under the control of these <u>FOR</u> clauses. The following NAME statement will supply all 400 values from the array COEF for printing:

```
NAME($ FOR I ← 1 STEP 1 UNTIL 40 DO $
   ($ FOR J ← 1 STEP 1 UNTIL 10 DO $
      (COEF[I,J])));
```

The "$" signs are necessary around a FOR clause when it is used as a replicator in a NAME (or PRINT) statement. Also, the phrase being replicated must be enclosed in parentheses, whether it is only a single expression like (COEF[I,J]) or a complex expression which itself contains a replicator, like:

```
($ FOR J ← ....DO $  (COEF[I,J]))
```

This accounts for the three sets of parentheses in the example above. The following is the syntax of a NAME statement:

```
<name statement> ::= NAME ( <name list> )
<name list> ::= <name list element> | <name list>, <name list element>
<name list element> ::= <name expression> | <replicator> ( <name list> )
<name expression> ::= <arithmetic expression> | <Boolean expression> |
    <logic expression>
```

This syntax shows that any simple or complex list of "names" may be enclosed in parentheses and replicated; such a replicated list may then be a single element in another list. The following legal name statement illustrates lists and replicated lists:

```
NAME ( A[1], $ FOR J ← 1 STEP 2 UNTIL 3 DO $
          (J, A[J], COEF[I,J]), A[7])
```

This example is equivalent to the following more simple statement:

```
NAME (A[1], 1, A[1], COEF[I,1], 3, A[3], COEF[I,3], A[7])
```

As another illustration, refer again to the example, where the row number is to be printed on every line of the matrix. The simplest way to print these numbers is to give their values in the NAME statement and use numeric instructions to place them into the print buffer. Thus, the following NAME statement will supply (in addition to the array value), the row number I just before the first element in each row:

```
NAME ($ FOR I ← 1 STEP 1 UNTIL 40 DO $
          (I, $ FOR J ← 1 STEP 1 UNTIL 10 DO $
                    (COEF[I,J])));
```

Since for clause replicators used in format programs very frequently start at one and increase in steps of one, an abbreviated notation has been provided for this special case. The replicator

<variable> → $ <arithmetic expression> $

has the same meaning as:

$FOR <variable> ← 1 STEP 1 UNTIL <arithmetic expression>$

Therefore, the NAME statement given above for the matrix with row numbers may be written more compactly as:

NAME (I → 40(I,J → 10(COEF[I,J]))) ;

One more simplification is possible in this form; in the special case that the <arithmetic expression> giving the upper limit of replication is a constant (like "40"), or a simple variable (like "N"), it need not be surrounded by "$" signs. Thus, for example, "I → N" is a correct replicator. "I → N-1" is incorrect since dollar signs are required around the arithmetic expression; the correct replicator would be "I → $N-1$".

To print the column headings in the example, the values 1, 2, ..., 10 must be supplied in a NAME statement. The simplest NAME statement for the column headings is:

NAME(I → 10(I)) ;

That is, I runs from 1 to 10, and it is the value of I itself which is to be printed.

The same forms of replicators which are used in NAME statements may also be used to execute repeatedly format programs or lists of format programs in PRINT statements. Thus, instead of writing "<2D, 2D, 2D, 2D>", we may write "J → 4 <2D>". In the case of a replicator in a PRINT statement, however, the actual value of the replicated variable frequently is not referred to; that is, the replicator is used simply as a counter. In such a case, the variable in a " → " replicator may be omitted; thus, " → 4 <2D>" may be used to get four repetitions of the format instruction "2D".

Following is the syntax of replicators:

<replicator> ::= $ <for clause> $ | <simple variable> → <limit> |
→ <limit>

<limit> ::= $ <arithmetic expression> $ | <simple variable> |
<unsigned integer>

Some examples of these forms follow:

```
$ FOR J ← 2, 3, K + 2 STEP 3 WHILE A[K] < K DO $
                J → $  (A[I] / 2) + 3 $
                J → N
                J → 3
                → $ (A[I] / 2) + 3 $
                → N
                → 3
```

If the upper limit of replication has a value such that zero or fewer replications are called for, then the phrase which is being replicated will be skipped entirely.

As an illustration, the NAME and the PRINT statement for printing the column heading of the example are:

```
NAME (I → 10(I));
PRINT (<16C>, → 10 <2D, 9R>, <E>);
```

Notice that the entire format program <2D, 9R> is replicated ten times. The replicators are <u>not</u> part of the format language, and must therefore appear <u>outside</u> the format brackets.

The variable I cannot be omitted from the replicator "I → 10" in the NAME statement, since I is referred to, and is, in fact, the value to be "named". It would definitely have been incorrect to have used the identical notation "I → 10" in our PRINT statement, since the same variable I is already being used for a different replicator in the NAME statement. Horrible confusion will result from using the same variable as a replicator at the same time in both a NAME statement and its parallel PRINT statement.

After the instructions "<16C>, → 10<2D, 9R>" have been executed, CP will be set to print position 126, past the RM of 120. However, this does not cause error printing because the two digits stored on the tenth

replication will be put into positions 115 and 116, and no attempt will be made to store characters in positions greater than RM.

Finally, we set up a PRINT statement for the matrix itself. Notice the extra blank line every five lines. To get this blank line, we need only execute an E instruction while the print buffer contains only blanks. Thus, our PRINT statement will have the form:

```
PRINT ( → 8(<E>, → 5 (format program for one line)))
```

The format program for one line could be:

```
<6C, 2D, 5R>, → 10 <-3D.4Z, 2R>
```

The entire program for the printed output of the example has now been developed:

```
PRINT (<P, 37C, 'ADJUSTED COEFFICIENT MATRIX  -  DELTA = '>);
NAME (DELTA) ; PRINT (<1D.1ZL, 4E>);
NAME (I → 10(I)); PRINT (<16C>, → 10 <2D, 9R>, <E> );
NAME (I → 40(I,J → 10(COEF [I,J])));
PRINT ( → 8(<E>, → 5 (<6C, 2D, 5R>, → 10 <-3D.4Z, 2R>, <E>)));
```

CHAPTER 3c

Introduction to READ

A useful way to visualize the process of reading alphanumeric information from cards is to consider READ to be the reverse process of PRINT. Recall that in printing, an image was formed in a buffer and then sent to the printer to be printed. In READ, however, the image originates at the input hardware and is then sent to an input buffer which is used by the READ statement in scanning string or numeric values; these are then loaded into variables named by a NAME statement. This buffer has a "CP", "LM", and "RM".

The NAME statement used with READ has the same form as with PRINT except that it supplies the names of variables rather than the values of the variables named. Therefore, the NAME statement used with READ forms a list of ALGOL variables (either simple or subscripted), not general arithmetic expressions, as are allowed with PRINT. Each numeric or alphanumeric instruction assigns a value to successive variables supplied by the NAME statement. Replicators may be used in the READ statement with the same meaning as in a PRINT statement.

The following sequence is incorrect:

NAME (A + B); READ (< 3D >)

since the NAME statement names an expression which is not a simple or subscripted variable.

The READ format program contains a list of instructions, very similar to those in PRINT, which control the reading of new cards and which specify the location and type of information expected to be found in the READ buffer. Thus, the programmer, by using suitable READ format instructions, is free to arrange his data cards in any format he desires.

The remainder of this chapter is divided into sections:

A. Control Instructions
B. Alphanumeric Instructions
C. Numeric Instructions
D. Card Overflow
E. An Example Using READ

It is assumed that the reader has read Chapter 3b.

A. Control Instructions

Just as the user uses E or W in a print format to control printing, so does he use E or W in read format to control reading.

nE Read n card images into the current READ buffer and set CP to LM. Only the last card image read is available after executing this instruction; hence, "1E" or "E" is the most common use of the instruction.

nW The action is the same as in "nE" except that the card images are also printed on the program listing.

In a READ format program, as opposed to a PRINT program, the E or W is usually the first instruction, rather than the last. The remainder of the format program then controls the scanning of the characters read into the read buffer. As in PRINT, the user has the ability to move CP:

nC	Set CP to column n.	CP ← n
nR	Move CP to the right n columns.	CP ← CP + n
nL	Move CP to the left n columns.	CP ← CP - n
nB	Equivalent to nR.	

B. Alphanumeric Instructions

As in printing, the user has the ability to input any string information with an nA instruction:

nA Scan the next n character positions of the read buffer and store the information there into ↓((n+3)/4) words from a NAME statement. The information is stored four characters per word, with the possible exception of the last word. If the last word does not get four characters, those characters it does get are stored right-justified.

As an example, assume that the characters 'ABCDE␣' appear on a card, with the 'A' in column 15. The effect of executing the statements

NAME(L, M); READ(<15C, 6A>)

will be to store 'ABCD' into L and '␣␣E␣' into M. CP will be left at 21.

Another possibility is to supply fixed string information directly from the READ statement, rather than from the card image. This ability is particularly useful in setting successive elements of an array to contain alphanumeric string information. We have

'<string>' The n characters between the quote marks are stored into ↓((n+3)/4) words from a NAME statement, just as for nA. CP is left unchanged.

Again, an example may be useful. Executing the statements

NAME(I → 5 (A[I])); READ(<'*THIS␣IS␣A␣STRING*'>)

is equivalent to executing

A[1] ← '*THI'; A[2] ← 'S␣IS'; A[3] ← '␣A␣S'; A[4] ← 'TRIN';
A[5] ← '␣␣G*'

The number of characters between the quotes is 18, not a multiple of four. Thus, the last two characters are stored right-justified in the fifth named variable.

The last alphanumeric instruction provides the ability to read Boolean values from a card.

nT The next n columns are scanned, but only the first non-blank column is examined. If it contains 'T', the corresponding name is set to true; otherwise, the corresponding name is set to false. If the corresponding name is not of type Boolean or logic, the error situation "ILLEGAL BOOLEAN" exists and will be treated as described below in Section D of Chapter 3d.

C. Numeric Instructions

Two essentially different methods are provided for reading numbers from cards: fixed field and free field. In the former, the programmer must specify (and therefore he must know) when he writes the program the columns on data cards in which the numbers will be punched. This format information is then part of the compiled program. With free field reading, the programmer specifies in his program only the number of quantities to be read. The numbers may then be punched in any format on the cards, separated by commas. Whether fixed field or free field is selected, however, the same rules govern the actual form of the numbers read. (The distinction between fixed field and free field only has to do with the columns used.) Numbers on data cards obey the same syntax as decimal numbers in program, with one addition: If a "/" is punched before the number, either before or after the sign, the number will be treated as an octal number. If an exponent appears, it will then be treated as an octal power of eight. (In summary: / on data cards is equivalent to 8F in program, but the latter notation is not allowed on data cards. 8L and 8R are also not allowed on data cards.)

Fixed field reading will be described first. For each number, the programmer may specify the following information:

1. Number of columns to be read.

2. Treatment of blank columns. Blanks may either be ignored or may be treated as if they were punched with a zero.

3. Decimal or octal conversion. The programmer may indicate that the number is to be read as an octal rather than a decimal quantity.

4. Scaling. The programmer may indicate that the value read is to be multiplied by a power of ten (or of eight for octal conversion).

5. Alarm suppression. Normally, reading a character other than a digit, +, -, decimal point, / or $_{10}$ will cause an alarm. However, the programmer may suppress this feature and cause such illegal characters to be ignored.

The syntax for a read numeric instruction is as follows:

<read numeric instruction> ::= <unsigned integer> D <read suffix>
<unsigned integer> Z <read suffix> | <int> F
<read suffix> ::= <empty> | <read suffix> <read suffix part>
<read suffix part> ::= H | N | E <integer>
<int> ::= <empty> | <unsigned integer>

The unsigned integer gives the number of columns to be scanned, and may be as large as 127. If D is used, blank columns are ignored, while using Z causes such columns to be treated as though they were punched with a zero. The suffix H causes the number to be treated as an octal quantity, regardless of whether or not a / is punched. A suffix of the form E±n causes the number read to be multiplied by ten (or eight) raised to the ±n power. The suffix N causes illegal symbols to be ignored.

Two error conditions may be detected in reading numbers: ILLEGAL SYMBOL and IMPROPER NUMBER. (A detailed description of error messages in READ is given in Section D of Chapter 3d.) The first indicates that a character other than a digit, +, -, decimal point, / or $_{10}$ has been read. It is this error message which is suppressed by the N suffix. The second message indicates that the number is improperly formed. For example, it may have more than one decimal point, more than one $_{10}$, a decimal point after a $_{10}$,etc.

In the numeric instructions just described, the field width or number of columns to be scanned is specified by "nD" or "nZ" and is fixed. A more flexible type of numeric instruction exists in the form of "nF" or free field read. "nF" specifies that n numbers are to be read and stored into the next n names. Each number field is terminated by a comma, thus allowing the data to be punched without reference to particular card columns. Numbers may be punched in the same forms as for the fixed-field READ and may continue from one card to the next. Blanks are ignored except that if an entire field is blank, the value of the corresponding name is not altered instead of being set to zero.

An "*" may be used in place of a comma to terminate a number field. This will stop the scanning of the card. If fewer than n numbers have been read, the remaining names will be left unaltered as though the corresponding number fields were left blank. For example, executing the statements

NAME(A, B, C, D, E, F); READ(<E, 6F>)

on the data card

12.6, /14_{10}+5, , 0 *

is equivalent to executing the statements

A ← 12.6; B ← 8F14_{10}+5; D ← 0;

It is clear, of course, that these statements leave C, E and F unaltered.

D. Card Overflow

If a READ statement attempts to scan past the right margin, a card overflow situation is said to exist. This situation is not treated as an error, but is taken care of automatically by the system. As soon as an attempt is made to read past the right margin, another card is read into the buffer using either an E or a W, depending which of these the user used last to read a card. CP is then set to LM (as usual), and the character is read from that column.

E. An Example using READ

To illustrate many of the concepts which have been discussed, a complete example follows, programmed in several ways. Assume an array A has been declared

<u>real</u> <u>array</u> A[1:80]

and that values for all 80 elements are to be read from cards. From the programmer's point of view, the simplest way to do this is the sequence

NAME(I → 80 (A[I])); READ(<E, 80F>)

Thus the numbers may be punched, as desired, on as many cards as needed, with successive numbers separated by commas. Assume instead that the data cards are already punched, without commas. Each card contains eight numbers, and each number is punched in nine columns with a column between

numbers whose contents are to be ignored. In this case, the READ statement given above might be replaced by

```
READ( → 10 ( <E>, → 8 <9D, 1R> ))
```

A more interesting possibility is the following: Suppose that the numbers are punched onto 80 cards and that each card has punched in columns 9 and 10 a subscript and between columns 12 and 30 a value. That is, the 80 cards may be placed in any order and the number in columns 9 and 10 indicates into which element of the array the value is to be stored. One way to program this is the following:

```
for i ← 1 step 1 until 80 do
begin NAME (j, A[j]);  READ(<E, 9C, 2D, 1R, 19D>) end
```

This sort of construction will work since the code for naming A[j] is not executed until after a value has been read into j. The reader should satisfy himself that the following will also work:

```
NAME( → 80(j, A[j] );  READ( → 80 <E, 9C, 2D, 1R, 19D>)
```

AL.3c.8

CHAPTER 3d

A Complete Description of ALGOL-20 Input/Output

A. Introduction

Chapter 3d is a complete, detailed description of input/output statements in ALGOL-20. This material is organized to be used for reference rather than for instruction. The user unfamiliar with the concepts involved should read first Chapters 3b and 3c which are primers on printing and reading, respectively.

Chapter 3d is divided into sections, as follows:

A. Introduction
B. NAME Statements and Replicators
C. PRINT and PUNCH Statements
D. READ Statements
E. Buffer Manipulations and "|" - variables
F. Control and Execution of I/O Statements

In the following, the term "format statement" will be used to refer to either a READ statement, a PRINT statement or a PUNCH statement, since the latter three types of statement are used to indicate the format of data. The term "output statement" will be used to refer to a PRINT statement or a PUNCH statement.

B. NAME Statements and Replicators

NAME statements are used to specify <u>values</u> to be output in a print or punch operation or to specify <u>locations</u> into which data is to be stored in a read operation. The NAME statement is not executed directly: instead it becomes active and functions as a list of values or locations which are evaluated when needed by a format statement. To clarify this concept, consider the program segment:

I ← 7; NAME(A[I]); I ← 12; PRINT(<3D, E>)

The value of A[12] will be printed -- not that of A[7].

Only one NAME statement may be active at any given time. If several NAME statements appear before a format statement, only the last executed NAME statement will be available to the format statement. Hence in the program segment:

NAME(A[1]); NAME(A[2]);
PRINT (<format list>)

only the one element, A[2], is available to the PRINT statement. This topic is discussed in detail in Section F, below.

A replicator is used in a NAME statement to indicate that an expression or list of expressions is to be used repeatedly. The replicator acts on the list of expressions in a manner analagous to a for statement acting on a statement in ALGOL. A replicator appears in one of three forms, the first of which is:

$ <for clause> $

This replicator causes the replicated name list to be used repeatedly until the for list is exhausted. An example is:

$ for I ← 1 step 1 until 3 do $ (A[I], B[I])

which is equivalent to

A[1], B[1], A[2], B[2], A[3], B[3]

The second form of replicator is:

<simple variable> → $ <arithmetic expression> $

This form is equivalent to

$ for <simple variable> ← 1 step 1 until <arithmetic expression> do $

with one important exception: The <arithmetic expression> is evaluated only once, when the first name is actually called for. If the arithmetic expression is a simple variable or an unsigned constant, the enclosing dollar signs may be omitted. For example:

I → N (J → $ 2*I $ (A[I, J]))

The third form of replicator is:

→ $ <arithmetic expression> $

This form functions in a manner similar to the one immediately above, except that the translator creates an internal counter to use in place of the simple variable. This form may be used whenever the controlled variable is not needed in the name list. As in the above form, the dollar signs may be omitted if the arithmetic expression is a simple variable or an unsigned integer. For example, the construction

I → N (→ I ('*'), A[I])

is equivalent to

'*', A[1], '*', '*', A[2], '*', '*', '*', A[3], ..., A[N]

Syntax for NAME Statements and Replicators

<name statement> ::= NAME (<name list>)

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

<name list element> ::= <name expression> | <replicator> (<name list>)

<name expression> ::= <arithmetic expression> | <Boolean expression> |
<logic expression>

<replicator> ::= $ <for clause> $ | <simple variable> → <limit> | → <limit>

<limit> ::= $ <arithmetic expression> $ | <simple variable> | <unsigned integer>

C. PRINT Statements and PUNCH Statements

Most of the instructions in an output statement serve to control the form and positioning of information as it is entered in the output buffer; hence, it is natural to discuss PRINT and PUNCH statements together. Because the

statements are so similar in function and in order to conserve memory locations, PRINT and PUNCH initially share a common output buffer. This means that storing characters with a PRINT statement alters any information which may have been stored by PUNCH statements, and vice-versa. In addition, PRINT and PUNCH share the same CP, LM, and RM, so that changing CP in PRINT changes it for PUNCH also. Initially CP and LM are set to 1, and RM is set to 120. Characters stored to the right of position 80 are ignored when executing an "E" or "W" instruction in PUNCH. Users may have independent buffers for PRINT and PUNCH by using the methods described in Section F of this chapter.

Instructions appearing in output statements fall into one of three classes: Control instructions to specify the position of information in the output buffer, alpha-numeric instructions to store constant information and alpha-numeric strings, and numeric instructions to specify the form in which numbers are to be stored.

Control Instructions

Associated with the output buffer are three variables: CP, LM and RM, the character pointer, the left margin and the right margin, respectively. CP points to the "next" position in the buffer into which information may be stored. LM and RM refer to the left-most and right-most positions in the buffer into which characters may be stored. The following instructions may be used to set or change CP or to output information:

nC Set CP to position n (column n). That is, CP ← n.

nR Move CP n positions to the right. That is, CP ← CP + n.

nL Move CP n positions to the left. That is, CP ← CP - n.
Moving CP to the left or right with nL or nR does not effect the contents of the positions in the buffer which are passed over.

nE Print (punch) one copy of the contents of the output buffer, output n - 1 blank lines (cards), clear the output buffer to blanks and set

CP to the left margin LM.

nW Print (punch) n identical copies of the output buffer on n successive lines (cards). The output is not cleared and CP is not moved.

P Upspace the paper to the top of the next page. (P is ignored in punch statements.) In general a message will be printed as the first line of the new page giving the date and a page number. The date is printed starting at the left margin in the form ' 04 JUL 64', and the page number is printed in the last ten columns before the right margin in the form 'PAGE nnnn ', where nnnn represents the number of pages printed since the end of compilation, in <4D> format. Printing of the page header is under the control of the programmer. He may restart the page numbering or suppress the header completely. See Section E below for details.

Executing "P" does not disturb the output buffer or CP. nP is treated as 1P or P, for any n.

In the above control instructions, and in the following alphanumeric instructions, n is assumed to be a positive, unsigned integer less than 512. If n is to be one, it may be omitted. For example, "E" is treated as "1E".

Alphanumeric Instructions

Alphanumeric instructions are used for all storage into the output buffer, except for storing of numbers. There is provision for storing strings which appear in the output statement, for storing quote marks, for storing alphanumeric information from a NAME statement, for storing blanks, and for storing Boolean quantities. Whenever a character is stored into the output buffer, it is stored into the position indicated by CP and CP is then incremented by one. However, before the storing is done, a check is made that LM $\leq$ CP $\leq$ RM. If this condition is not met, an "E" is executed and the character is then stored at the LM of the next line.

'<string>' The characters of the string appearing between the quote marks are stored. Any G-20 character except quote may be stored by this instruction.

nQ n quote marks are stored.

nA n alphanumeric characters are stored. These characters come from ↓((n+3)/4) names from a NAME statement. Each name, with the possible exception of the last, supplies four characters to be stored. The characters from the last name are taken from the right end of the word.

An example of an A primary may help. Assume that A[1] and A[2] have been named, containing 'STRI' and '**NG' respectively. Executing <6A> will cause 'STRING' to be stored into the output buffer. Had <7A> been executed instead, 'STRI*NG' would have been stored.

nB n blanks are stored. nB has the same effect as a string instruction with n blanks between the quotes.

nT A Boolean value is stored. The number of characters stored into the output buffer is min(5, n). The characters stored are taken from one of three strings, depending on the value, v, of the next NAME. If v is true, the string used is 'TRUE␣'; if v is false the string is 'FALSE'; and in all other cases the string is 'UNDEF'. (The latter may occur if the NAME is not a Boolean quantity.) The two most useful forms of this instruction are 1T, which stores 'T', 'F' or 'U', and 5T, which stores 'TRUE␣', 'FALSE' or 'UNDEF'.

Numeric Instructions

Numeric instructions are those instructions used to store numbers into the output buffer. Such an instruction may be regarded as giving a "picture" of the number to be stored. It includes the following information, some of which may be omitted if not needed:

(1) Sign control: The sign may be omitted or it may be stored. If the

latter, two more choices are available: Positive numbers may or may not have an explicit plus sign, and the sign may be either left-justified in the field or it may appear just before the left-most digit.

(2) Dollar control: Numbers may be stored as dollar amounts, with the dollar sign either left-justified or just before the left-most printed digit.

(3) Digits to the left of the decimal point: The number of such digits, if any, is specified. Leading zeros may be replaced by blanks.

(4) Decimal point: The decimal point may or may not appear, although if (5) is used it *must* appear.

(5) Digits to the right of the decimal point: The number of such digits, if any, is specified. Trailing zeros may be replaced by blanks.

(6) Exponent part: Several forms of "floating point" notation are available.

(7) Miscellaneous - the user may specify four more options: The number may be stored decimal or octal; special spacings may be used; alarm output may be suppressed; and the number may be truncated rather than rounded.

The syntax of a numeric instruction is as follows:

<numeric instruction> ::= <prefix> <number form> <suffix>

The prefix contains the specification of items (1) and (2); the number form contains the specification of items (3), (4) and (5); and the suffix contains the specification of items (6) and (7).

Consider first the number form, with the following syntax:

<number form> ::= <integer part> | <integer part> . |
<integer part> . <fractional part> | . <fractional part>
<integer part> ::= <unsigned integer> D | <unsigned integer> Z
<fractional part> ::= <unsigned integer> D | <unsigned integer> Z

Let the integer part be of the form νD or νZ, and the fractional part be of the form ηD or ηZ. If the integer (fractional) part is missing, let ν (η) be zero. Then the number will be stored with ν digits to the left of the decimal point and η digits to the right. If the integer (fractional) part contains a D, leading (trailing) zeros will be replaced by blanks, while the Z form causes such zeros to be stored. If ν (η) is zero, then no digits will

be stored to the left (right) of the decimal point. If ν (η) is non-zero, at least one non-blank character will be stored to the left (right) of the decimal point, even though a zero must be stored where D format would otherwise indicate a blank. The decimal point is stored whenever it is present in the number form. The number is normally rounded by adding five to the first digit to the right of the last digit stored. The sum of ν and η must be less than 15.

The prefix is the specification of sign and dollar sign. The syntax of the prefix is as follows:

<prefix> ::= <$ part> <sign part> | <sign part> <$ part>
<$ part> ::= <empty> | L$ | $
<sign part> ::= <empty> | L+ | L- | + | -

In both the sign part and the $ part, the presence of "L" indicates left-justified. A sign or dollar sign specified by "L+", "L-" or "L$" will be stored into the output buffer before any digits or blanks, while a sign or dollar sign specified by "+", "-" or "$" will be stored just before the first non-blank digit stored by the number form. The order of storing is as follows:

1. $ specified by "L$"
2. sign specified by "L+" or "L-"
3. blanks from suppressed leading zeros in D-type integer part
4. sign specified by "+" or "-"
5. $ specified by "$"
6. first non-blank character from number form

The sign part specifies one of three possible formats for storing the sign of the number. If it is empty, no sign is stored, even though the number may be negative. If it is "+" or "L+", a sign, either '+' or '-', will be stored, taking one space. If it is "-" or "L-", a '-' will be stored if the number is negative and a blank will be stored otherwise.

The suffix part of a numeric instruction is used to supply supplementary information: scaling of the number, storing the exponent, special spacing and other options. The syntax is as follows:

<suffix> ::= <empty> | <suffix> <suffix element>

<suffix element> ::= L | S <integer> | F <integer> | E <integer> | H | K | N | T

The various suffix elements are explained below. If an exponent is stored, it takes six positions in the output buffer, in the form: '␣$_{10}$±dd␣'. No more than one of the suffix elements S, L, F or E should be used on a given suffix.

L The number is left-justified in the field specified by the number form, and the resultant exponent is stored. This is "scientific notation".

E±n The number is shifted so that its exponent equals ±n and the exponent is stored.

F±n The number is shifted so that its exponent equals ±n, but the exponent is *not* stored.

S±n The number form portion of a numeric instruction containing this suffix must be of the form "<integer part> . ". (The decimal point must appear.) The number is shifted so that its exponent equals ±n. The resultant mantissa is then left-justified in the specified field. The two shifting operations determine the position of the decimal point, which is then inserted where needed. The resulting exponent is stored.

H The number is stored octal rather than decimal. If an exponent is stored, it is to be interpreted as a power of eight.

K One of two special spacings is used in storing the number. If a $ part appears in the prefix, the digits of the number are stored in groups of three, separated by commas. If a $ part does not appear, the digits are stored in groups of five separated by spaces. In either case, the groups are counted left and right from the decimal point. The decimal point, if present, serves as one of the spaces.

N Possible alarm output is suppressed (see the text below), and any digits which overflow the left end of the field are lost.

T The number is truncated after the last stored digit, rather than rounded as usual.

If any format other than L is used, it is possible that the magnitude of the number is such that there are more digits to the left of the decimal point than can be stored using the specified number form. In such a case (providing that the suffix "N" was not used), alarm output will take place with the use of "L" format. If E or S format was called for, no extra spaces will be taken. Otherwise the number will take six more spaces than expected. For example, the number 123 will be stored as 12 $_{10}$+01 by 2D, but as 23 by 2DN.

Examples of Numeric Instructions

The value of the number to be stored is 4673900. The numeric instructions listed on the left side of the page will cause the storing; of the corresponding strings of characters. The numbers at the right indicate the numbers of buffer positions used.

Instruction	Characters stored	Positions
7D	4 6 7 3 9 0 0	7
8D	␣ 4 6 7 3 9 0 0	8
9Z	0 0 4 6 7 3 9 0 0	9
7D.4D	4 6 7 3 9 0 0 . 0 ␣ ␣ ␣	12
7D.4Z	4 6 7 3 9 0 0 . 0 0 0 0	12
8DL	4 6 7 3 9 0 0 0 ␣ $_{10}$ - 0 1 ␣	14
3D.1DL	4 6 7 . 4 ␣ $_{10}$ + 0 4 ␣	11
1Z.2ZE+7	0 . 4 7 ␣ $_{10}$ + 0 7 ␣	10
1Z.2ZF+7	0 . 4 7	4
3D.3ZF+7	␣ ␣ 0 . 4 6 7	7
.3ZF+7	. 4 6 7	4
+7D	+ 4 6 7 3 9 0 0	8
-7D	␣ 4 6 7 3 9 0 0	8
L$+8D	$ ␣ + 4 6 7 3 9 0 0	10
L$-8D	$ ␣ ␣ 4 6 7 3 9 0 0	10
$+8D	␣ + $ 4 6 7 3 9 0 0	10
8Z.K	0 4 6 ␣ 7 3 9 0 0 .	10
L$8D.2ZK	$ ␣ 4 , 6 7 3 , 9 0 0 . 0 0	14
3D.1DF+4	4 6 7 . 4	5
3D.1DF+4T	4 6 7 . 3	5
8Z.S+3	4 6 7 3 . 9 0 0 0 ␣ $_{10}$ + 0 3 ␣	15
8Z.S+4	4 6 7 . 3 9 0 0 0 ␣ $_{10}$ + 0 4 ␣	15
4DN	3 9 0 0	4
4D	4 6 7 4 ␣ $_{10}$ + 0 3 ␣	10 *
8DH	2 1 6 5 0 5 5 4	8
4D.2ZHL	2 1 6 5 . 0 6 ␣ $_{10}$ + 0 4 ␣	13
3ZHNTF+3	6 5 0	3

* Alarm output used.

Syntax for Print and Punch

For the purpose of this syntax, the G-20 characters "<" and ">" will be replaced by "≪" and "≫" , respectively. "<" and ">" will be reserved for meta-linguistic brackets in the Backus Naur Form syntax.

```
<print statement> ::= PRINT ( <format list> )
<punch statement> ::= PUNCH ( <format list> )
<format list> ::= <format list element> | <format list> , <format list element>
<format list element> ::= ≪ <format program> ≫ |
          <replicator> ≪ <format program> ≫ | <replicator> ( <format list> )
<format program> ::= <format instruction> |
          <format program> , <format instruction>
<format instruction> ::= <control instruction> | <alphanumeric instruction> |
          <numeric instruction>
<control instruction> ::= <int> C | <int> R | <int> L | <int> E | <int> W |
          <int> P
<alphanumeric instruction> ::= <string> | <int> B | <int> Q | <int> A |
          <int> T
<numeric instruction> ::= <prefix> <number form> <suffix>
<prefix> ::= <$ part> <sign part> | <sign part> <$ part>
<sign part> ::= <empty> | L+ | L- | + | -
<$ part> ::= <empty> | L$ | $
<numeric primary> ::= <integer part> | <integer part> . |
          <integer part> . <fractional part> | . <fractional part>
<integer part> ::= <unsigned integer> D | <unsigned integer> Z
<fractional part> ::= <unsigned integer> D | <unsigned integer> Z
<suffix> ::= <empty> | <suffix> <suffix element>
<suffix element> ::= L | H | N | K | T | S <integer> | E <integer> |
          F <integer>
<unsigned integer> ::= <digit> | <unsigned integer> <digit>
<integer> ::= <unsigned integer> | + <unsigned integer> | - <unsigned integer>
<int> ::= <empty> | <unsigned integer>
<string> ::= ' <proper string> '
<proper string> ::= <empty> |
          <proper sting> <any G-20 character other than quote>
```

Execution of Print and Punch Statements

From the definitions of PRINT and PUNCH statements, it is evident that the forms of these statements are:

```
PRINT  ( fle, fle, ..., fle )
PUNCH  ( fle, fle, ..., fle )
```

where "fle" denotes a format list element. The fle's are executed in order of appearance, from left to right. After the rightmost fle is executed, the statement is terminated. Each fle is either a format program bracketed by "< >" and possibly replicated, or a replicated list of fle's, separated by commas. In turn each format program may be a list of format instructions (Eg., "3C, 2Q, E"). These instructions are also executed in left to right order. It should also be noted that *no replicators may appear inside the "<" ">" brackets*. If a format instruction requires a value, it will cause a call on the corresponding NAME statement and evaluate the next expression to obtain a value.

D. READ Statements

Most instructions in a READ statements are used to scan data which has been read into an input buffer and to store data values into variables which have been named in a NAME statement. As in PRINT and PUNCH, the instructions fall into three classes: *control instructions* to control the reading of data card images into the buffer and the positioning of CP, *alphanumeric instructions* to specify the manner in which alphanumeric data is to be scanned and stored into variables, and *numeric instructions* to specify the manner in which numbers are to be scanned, interpreted and stored into variables.

Control Instructions

Associated with the input buffer are three variables: CP, LM, and RM -- the Character Pointer, the Left Margin, and the Right Margin. CP points to the "next" position in the buffer which is to be scanned. LM and RM refer to the left-most and right-most positions in the buffer which may be scanned. The following instructions may be used to set or change CP:

nC	Set CP to position n (Column n). That is, CP ← n.
nL	Move CP n positions to the Left. That is, CP ← CP - n.
nR	Move CP n positions to the Right. That is, CP ← CP + n.
nB	Equivalent to "nR".

The following two instructions may be used to read data card images into the input buffer:

nE	Read n card images into the current READ buffer, and set CP to LM. At the completion of this instruction, only the last card image read is available to be scanned.
nW	The action is as in "nE", except that the card images are also printed on the program listing.

In the above control instructions, and in the following alphanumeric instructions, n is assumed to be a positive, unsigned integer less than 512. If n is one, it may be omitted. For example, "W" is treated ad "1W".

Alphanumeric Instructions

Alphanumeric instructions are used to scan alphanumeric characters and store string or Boolean values into variables named in a NAME statement:

nA The next n character positions of the input buffer are scanned, and the string of n characters there is stored, four characters per word, into the next ↓((n + 3)/4) named variables. If n is not a multiple of four, the

characters stored in the last variable are right-justified.

'<string>' — The n characters of the string are stored as in "nA". CP is not changed.

nT — The next n character positions are scanned and a Boolean value is stored in the next variable named. If the first non-blank character scanned is the letter "T", the value of the variable is set to TRUE; otherwise, it is set to FALSE. CP is incremented by n. If the variable named is not of type Boolean or logic, the error condition "ILLEGAL BOOLEAN" is detected and treated as described below.

Numeric Instructions

READ numeric instructions are either fixed-field of free-field. Fixed-field instructions consist of a primary specifying field width (the number of characters to be scanned) and possibly a suffix specifying additional information, such as scaling or octal conversion.

nD (nZ) — The instructions "nD" and "nZ" are used to form READ primaries. "nD" scans the next n character positions of the buffer for a real or integer number and stores it in the corresponding name. Any blanks scanned are ignored, with the exception that if the entire field of n character positions is blank, the value zero is stored. A number preceded by a "/" is treated as an octal (base eight) number. n must be a positive integer less than 128. The instruction "nZ" functions as "nD" except that blanks are treated as zeros. The forms "nD.", "nD.nD" ".nD" and the corresponding Z primaries are not correct in READ.

The suffix of the fixed-field instruction may be empty or may consist of one or more of the following suffix parts:

H — The number is converted in octal (base eight) regardless of whether or not it is preceded by a "/". If the number has an exponent, the exponent is treated as a power of eight.

E±n — The number read is multiplied by ten (or eight) to the power ±n.

N — Any character other than a digit, +, -, decimal point, /, or $_{10}$ is ignored if it is scanned. CP is incremented by one, and the next character is scanned. Normally, scanning any character other than those listed above will result in the detection of the error condition "ILLEGAL SYMBOL".

In the numeric instructions just described, the field width or number of columns to be scanned is specified by "nD" or "nZ" and is fixed. A more flexible type of numeric instruction exists in the form of "nF" or free read:

nF — n numbers are to be scanned and stored into the next n variables named. Numbers may be punched in the same forms as for fixed-field read, and each number field is terminated by a "," or a "*". Blanks are ignored, except that if an entire field is blank, the value of the corresponding variable is left unaltered instead of being set to zero.

A "*" terminates the scanning of the "nF" instruction. If fewer than n numbers have been scanned, the values of the remaining variables named are left unaltered, as though the corresponding number fields were left blank. After execution of "nF", CP points to the character position one position to the right of the last "," or "*" scanned.

Card Overflow

If a READ instruction attempts to scan character positions past the right margin, a new card image is read using a pseudo control instruction. This instruction functions as an "E" or "W" instruction, whichever has been executed most recently. Scanning continues with CP set to LM. Initially, CP = 1, LM = 1, and RM = 84.

Error Messages

Several situations are detected by the input routine as indicating an error by the user, either in his ALGOL I/O call or in his data cards. A standard error printout is provided, containing the following information:

1. The last card read is printed. (If it was read by a W, it will thus be printed twice.) The next line will contain an integer giving the present value of CP and will also have a vertical arrow (↑) pointing to the column indicated by CP. Usually, this will be the column just past the error.

2. A single line is printed identifying the particular error.

3. The standard ALGOL run error mechanism is invoked with RUN ERROR - READ. The following error messages (item 2, above) are detected:

ILLEGAL BOOLEAN An attempt has been made to read with a T instruction into a variable of type other than Boolean or logic.

$$ - CARD READ An attempt has been made to read past an end-of-file mark. Reading more card images than are in the current input file results in reading an end-of-file mark. This mark consists of special dollar signs (internal representation 165_8) in columns one and two. Attempting to read still another card image causes the error condition "$$ CARD READ" to be detected.

NO CARD READ An attempt has been made to scan information before an E or W instruction has loaded the input buffer.

IMPROPER NUMBER In scanning a number with a numeric instruction, an illegal sequence such as more than one decimal point, more than one $_{10}$, or a decimal point after a $_{10}$ has been detected.

ILLEGAL SYMBOL In scanning a number with a numeric instruction, a character other than a digit, +, -, decimal point, / or $_{10}$ has been read. This message is suppressed by the suffix N.

E. Buffer Manipulation and | - variables

As has been mentioned, an input buffer and an output buffer exist in the I/O system. Associated with each buffer are three pointers: CP, LM and RM. It is frequently convenient for the programmer to be able to make direct reference to these buffers instead of being restricted to using format instructions to refer to them. For example, in all that has been said up to this point no mention has been made of any way the programmer can change LM or RM. To permit reference to the various pointers of the I/O system, ALGOL-20 includes a special class of reserved words: the bar-variables. These variables consist of a vertical bar ("|") followed by an integer. The | and the first digit of the integer must be in successive columns of the same card, with no intervening blanks.

The format of a buffer will now be described using the print buffer for definiteness. The buffer itself consists of 120 consecutive locations in memory, corresponding to the 120 columns of the printer. Characters are stored into the buffer by placing the G-20 representation of each character in the corresponding word, right-justified. The three pointers associated with the buffer are stored in the three locations immediately before that containing column one. "Column zero" contains CP, "column -1" contains RM and "column -2" contains LM. Each of these three pointers has a name which is available to the user, the name being a bar-variable. For the print buffer, CP is in |205, RM is in |206 and LM is in |207. Thus the assignment statement

|205 ← 5

is equivalent to the format statement

PRINT(<5C>)

Similarly, the programmer may change the right margin by storing into |206 with an assignment statement.

A similar situation exists for the input buffer. 84 consecutive locations are provided for the actual read buffer. Column zero, called |200, contains CP for reading; column -1, |201, contains the read RM; and column -2, |202, contains the read LM.

Since PRINT and PUNCH share a common buffer, it follows that they share a common CP, RM and LM.

The following table may help to clarify the preceding discussion:

Location	Initial Contents	Meaning	
|202	1	LM	READ
|201	84	RM	
|200	1	CP	
next 84 words	--	the buffer	
|207	1	LM	PRINT and PUNCH
|206	120	RM	
|205	1	CP	
next 120 words	--	the buffer	

This gives the programmer convenient access to the three pointers, but it does not provide a way to refer to the words in the buffer. Since it is frequently desirable for the user to have this ability, a means has been provided for the user to cause a buffer to be in his own data area instead of in the I/O system. Again considering PRINT, the user may direct that a particular 123 element array is to be used as the buffer. The system will then use the first three locations of this array as the three pointers and the other 120 locations as the print buffer. Since the array is in the user's memory, he may refer to any column or to any pointer by the ALGOL name he has given it. For example, assume that the declaration

logic array BUFF[-2 : 120]

has been used and that the procedure call

BUFFER.SET ('PRINT', BUFF[0])

has been executed. (BUFFER.SET is a privileged identifier.) Then for any k between one and 120, column k will be in BUFF[k]. CP will be in BUFF[0], RM will be in BUFF[-1] and LM will be in BUFF[-2]. It is important to note that |205, |206 and |207 are specific machine locations and that after executing the above BUFFER.SET call they will no longer contain the pointers.

BUFFER.SET may also be used to change the READ or PUNCH buffer, using the string 'READ' or 'PUNCH' as the first parameter to the procedure. As for PRINT, the second parameter should be an array element which will be set to correspond to "column zero" of the buffer.

Before calling BUFFER.SET, the programmer should be sure that the three pointers he is about to put into effect contain reasonable values. BUFFER.SET only makes one check: it insists that the relationship

$$0 < LM < RM$$

be satisfied. If it is not, LM will be set to one and RM will be set to 84, 120 or 80 for READ, PRINT or PUNCH, respectively.

BUFFER.SET detects two error conditions which are treated as run errors: a first parameter which is not one of the three legal strings allowed, or a second parameter which is not in the user's memory.

There are certain other bar variables associated with the input/output system which are available to the user. |210 and |211 are switches for format and NAME, respectively. At any given time during the running of a program when the user has NAMEd variables which have not yet been printed, |211 will be non-zero. (Its value is the location of a routine which will supply the names to succeeding statements.) If the programmer wishes to cancel the effect of the extra names which have been supplied, he may do so by setting |211 to zero. Similarly, extra format elements which have been supplied may be cancelled by setting |210 to zero. The programmer should under no circumstances set either of these variables to non-zero values, or chaos will result.

|212 and |213 are associated with the message printed at the top of each

page. Whenever the printer is moved to the top of a new page by the execution of a P, the user may have a message and page number printed if he so chooses. The system has been set so that the page numbers will start with page one on the first after the completion of the compilation. If the user does nothing about it, each time a P is executed the first line of the new page will contain on the left the date on which the program was run, and on the right the page number. The page number is calculated by finding out from the monitor the total number of pages which have been printed since the run began and subtracting from this number the contents of |212. The contents of |212 is set on entry to the program to the number of pages used by the compiler in compiling the program. The user may change it at any time if he wishes to alter the page numbering sequence.

|213 controls the message to be printed as part of the page header. If it is negative, no page heading at all will be printed. If it is zero, the date and page number will be printed, as explained above. Positive values should not be used in this location. In the present version $|213 > 0$ will be treated as suppressing the header, but in planned expansion it will have a different meaning.

|214 is the up-space counter. After each line is printed, the printer is up-spaced the number of lines indicated by |214. This location is set on entry to the program to one, for single spacing. The user may set it to two for double spacing, but other values are not recommended. In particular, setting it to zero saves paper but makes it hard to read the output.

|215 is the left-justify switch. In processing number forms there are certain occasions when either blanks or zeroes will be stored depending on whether the programmer has used D or Z in his format. If a blank would have been stored and if, further, |215 is zero, then no space will be taken in the print line instead of leaving a blank. Thus setting |215 to zero permits the user to get left-justified numbers. |215 is initialized to be non-zero.

These last few bar-variables may be summarized as follows:

|210 NAME switch. $\neq 0 \Rightarrow$ there are names to be processed

|211 format switch. $\neq 0 \Rightarrow$ there are formats to be processed

|212 page count

|213 page header switch. $< 0 \Rightarrow$ suppress; $= 0 \Rightarrow$ print; > 0 (do not use)

|214 upspace counter

|215 left-justify switch. $= 0 \Rightarrow$ left-justify; $\neq 0 \Rightarrow$ don't

F. Control and Execution of I/O Statements

The relationship between NAME and format statements is given in the following description of the execution of an input/output operation.

(1) An execution of a NAME statement sets the name switch, |211, to a positive integer, and sets an internal variable δ to point to the first name expression. Whenever |211 is positive the NAME statement which set it so is said to be active. A NAME statement becomes active as encountered, cancelling any previously active name statement.

When a NAME statement becomes active, a test is made to determine if a format statement is already active ($|210 > 0$). If no format statement is active ($|210 = 0$), control passes to the successor of the NAME statement. If a format statement is active, the first name expression is evaluated and sent to the format instruction pointed to by γ. (See (2).) δ is changed to point to the next name expression, and control passes to the active format statement.

(2) An execution of a format statement sets the format switch, |210, to a positive integer. The format statement is then said to be active. Because PRINT, PUNCH, and READ statements share the switch, at most one format statement may be active at any given time. A format statement becomes active when encountered, cancelling any previously active format statement.

The value (address) of a name expression may be needed during the execution of a format statement. If so, an internal variable, γ, is set to point to the format instruction requesting the value (address), and a test is made to determine if a NAME statement is active ($|211 > 0$). If not ($|211 = 0$), control passes to the successor of the format statement. If a NAME statement is active, control passes to the expression pointed to by δ.

(3) In attempting to evaluate a name expression, a check is made to determine whether δ points to an expression or to the end of the NAME

statement. If δ points to an expression, it is evaluated and δ is set to point to the next name expression (or to the end of the NAME statement), and the value (address) of the expression is sent back to the requesting format instruction. When all name expressions have been evaluated, δ points to the end of the NAME statement. In this case, no expression can be evaluated, and |211 is set to zero indicating that no NAME statement is active. Control is passed to the common successor of the now inactive NAME statement and the active format statement. (See (5).)

(4) After the last format instruction in a format statement is executed, |210 is set to zero, and control is passed to the common successor. (See (5).)

(5) The common successor of an active statement and a statement which has just become inactive is the successor of that statement which was most recently encountered during the execution of the ALGOL program.

To clarify the above points, consider some examples of sequences of input/output operations. In the following, N(P) denotes a NAME statement with P name expressions, F(P) denotes a format statement (PRINT, PUNCH or READ) which requires P values or addresses, S denotes an arbitrary ALGOL statement, and S' denotes any ALGOL statement which is not an input/output statement.

A: N(6); S'; F(6); S;

Executing N(6) sets |211 > 0 and sets δ to point to the first name expression. S' is executed and eventually F(6) is entered. Because N(6) is already active, each request for a value or address will be filled by N(6). When the execution of the last format instruction is complete, F(6) becomes inactive, and S is executed. N(6) is still active, but δ points to the end of statement. In this state, any request for a name expression will render N(6) immediately inactive. A NAME statement followed by a format statement is the simplest and most frequently used sequence.

B: F(5); S'; N(5); S;

B illustrates an alternate sequence, in which the format statement precedes the NAME statement. Executing F(5) sets |210 > 0, but no requests for name expressions can be filled because there is no active NAME statement. γ is set to point to the first requesting format instruction, and S' is executed. When N(5) becomes active, it determines that F(5) is already active.

The first name expression is evaluated and sent to the format instruction indicated by γ. Eventually, the last format instruction in F(5) is executed and F(5) becomes inactive. As in example A, N(5) is still active but any request for a name expression will render it inactive. Control then passes to S.

C: N(4); N(2); F(3); S'; N(1); S;

C illustrates a more complex situation which is probably a programming error. N(4) becomes active, but is cancelled by N(2). N(2) and F(3) function as in example A, except that when F(3) requests a third name expression, N(2) becomes inactive. S' is executed and N(1) encountered. N(1) now supplies the requested name expression to F(3) and F(3) becomes inactive, passing control to S. Users should be wary about using sequences such as described in C as it is very easy to produce an error which has repercussions on many other input/output operations in the program. As a safeguard, the name and format switches may be zeroed as described in Section E of Chapter 3d.

AL.3d.24

CHAPTER 4

SYSTEM STATEMENTS

System statements are instructions to the ALGOL-20 translator which may be used to modify certain aspects of the translation process. That is, a system statement is executed by the translator at compile time rather than by the object program at execution time. System statements are executed as they are encountered by the translator as it scans once through the ALGOL source program and take effect immediately thereafter. All system statements except those marked with "†" may be used anywhere in the source program.

Each system statement is punched on a separate card which contains "SY" in columns 1 and 2. The system statement itself may be punched on the card anywhere between column 4 and the current right margin (see RIGHT MARGIN below).

A system statement generally has the form:

< statement name > < parameters >

Those system statements which have a fixed number of parameters are terminated by a blank following the last parameter. The rest of the card may be used for comments. The system statements which have a variable number of parameters are terminated by the end of the card, so comments cannot be included on such a card. System statements which may not contain a comment on the same card are marked with a dollar sign ($).

Each system statement type will now be described and explained. In the following, "n" will always stand for an unsigned integer. The system statements marked with asterisks (*) control printing but will never themselves be printed. Printing of the other system statements may be suppressed by the system statement: "PRINT NO SYSTEM".

Blanks are *not* ignored when scanning system statements. There must be at least one blank *between* words and/or numbers and none *in* words or numbers.

PRINT CONTROL

(1) PAGE (No parameters)

The effect is to skip the compilation listing to the top of the next page. The PAGE statement itself will be printed on the first line of the new page.

*(2) LINE n

The effect is to upspace the printer by n lines. An attempt to upspace beyond the bottom of the current page will leave the paper at the top of the next page.

*(3) SINGLE (No parameters)

*(4) DOUBLE (No parameters)

These statements cause the compilation listing which follows to be printed with single or double line spacing, respectively. If neither statement is given, the translator assumes SINGLE.

*(5) INDENT n $K \leftarrow n$

INDENT +n $K \leftarrow K + n$

INDENT -n $K \leftarrow K - n$

The indentation constant, K, specifies the number of print positions to the right of the text left margin that the compilation listing will be printed. The translator normally assumes "IDENT 0". An IDENT card modifies K as given above. If this rule leaves K outside of the range $0 \leqq K \leqq 21$, then $K \leftarrow \max(0, \min(K,21))$. Note the difference between "IDENT 2" and "IDENT +2": The former sets K to 2 and the latter increments K by 2. See Chapter 6c for a discussion of the format of the compilation listing.

$(6) PRINT

The user has the ability to turn on or off the printing of various aspects of his source program. In general, if he does not specify otherwise, his source program along with octal addresses, notes from the translator, and system statements will be printed, while routines accessed from the symbolic library will not. The

* Not printed
$ Comment not allowed

printing of each of ADDRESSES, NOTES, SYSTEM statements and LIBRARY routines may be controlled individually by the programmer by suitable PRINT statements. We have the following syntax:

<PRINT statement> ::= PRINT <parameter string>
<parameter string> ::= <parameter>, | <parameter string> <parameter>,
<parameter> ::= <control word> | NO <control word> | NO | EACH
<control word> ::= PROGRAM | ADDRESSES | NOTES | SYSTEM | LIBRARY

A PRINT statement is interpreted by treating each of the parameters in the parameter string in order from left to right across the card. The control word ADDRESSES refers to the octal addresses printed down the left side of the page. NOTES refers to possible error notes printed by the translator. (See Chapter 6b.) SYSTEM refers to system statements (except those which never print). LIBRARY refers to routines accessed from the symbolic library, as described below. PROGRAM refers to the listing of the source statements, along with notes, addresses and associated system statements.

A parameter consisting solely of a control word has the effect of turning on the printing of the corresponding part of the assembly listing as described above, while a parameter of NO followed by a control word turns off that part.

The parameter EACH is equivalent to the parameter string "PROGRAM, ADDRESSES, NOTES, SYSTEM, LIBRARY", and the parameter NO is equivalent to "NO PROGRAM, NO LIBRARY".

The parameter NO PROGRAM suppresses printing of the source program along with the associated addresses, notes and system statements, overriding any previous parameter of ADDRESSES, NOTES or SYSTEM. If PRINT PROGRAM is in effect, however, then NO NOTES, NO ADDRESSES or NO SYSTEM will suppress printing of these individual features. It is not possible to print notes, addresses or system statements without printing the corresponding source program images. If PRINT NO PROGRAM is in effect, PAGE and LINE have no meaning and thus are ignored by the translator.

The parameter LIBRARY has a function analogous to PROGRAM, except that LIBRARY takes effect only when a subsequent SY LIBRARY system

statement starts inserting library images; then, having PRINT LIBRARY (PRINT NO LIBRARY) in the "main" program text has the same effect as having PRINT PROGRAM (PRINT NO PROGRAM) as the first library image. These library images may themselves contain PRINT system statements; these will control printing only within the library segment, so that the PRINT status in effect when the SY LIBRARY statement was encountered will be restored at the end of the LIBRARY segment. If SY LIBRARY occurs within a set of library images, print control works, as described above, calling the outer set of library images the main program and the inner set the library images. The PRINT parameters are always "pushed down" when an SY LIBRARY system statement is encountered, and the "LIBRARY" switch on one level becomes "PROGRAM" switch on the next level.

In the absence of any PRINT system statements, the ALGOL-20 translator assumes PRINT EACH, NO LIBRARY.

MISCELLANEOUS

(7) RIGHT MARGIN n

Starting on the next card, the translator will scan column 4 through n for the text of ALGOL, WHAT, and system statements, where $40 \leqq n \leqq 80$. If n is not in the proper range, an error message is given and the right margin is not changed. The translator initially assumes RIGHT MARGIN 72.

(8) LIBRARY <identifier>

The translator inserts into the program at this point the segment of ALGOL source program text (generally a procedure) which is filed in the symbolic library under the name <identifier>.

†(9) n ABCONS

The translator will reserve n G-20 locations for storing "abcons" and n locations for storing "adcons" during both translation and execution. Abcons are numbers and alpha-numeric strings which do not appear in format primaries. Adcons are constants and temps

† Before the first begin only

for format replicators and small integer constants used as actual parameters to procedures.

If no ABCON statement is given, the translator assumes "200 ABCONS".

An ABCON statement may only occur at the very beginning of the program before the first begin.

(10) SEGMENT n1, n2

The integer n1 specifies the segment number, and must usually satisfy $1 \leqq n1 \leqq N$. Segments 1 through N are temporary segment and thus are not saved after the end of the user's run; permanent segments with numbers greater than N, are available upon request to the Computation Center. Since the number of temporary segments may change in the future, no value for N is given here. Its value can be found in the Users Manual, Section 1.5. The integer n2 specifies the number of files which will be required for the segment; each file contains 10240_{10} words. If segment 1 requires 2 files, the next available segment is segment 3.

The number of "words" printed out at the end of an Algol program is the number that must be dumped out if the program is dumped as a segment.

See Chapter 6f for a complete discussion of segments.

(11) RELEASE WHAT

(12) RELEASE SYMBOLIC LIBRARY

The user who does not need WHAT or the symbolic library may reclaim the space used by these parts of the ALGOL compiler. This allows longer programs to be compiled. Since WHAT is below the library processor in memory, no space can be reclaimed until WHAT is released. The RELEASE's may be done, however, in either order and at any time during the compilation. Releasing WHAT reclaims /1400 (768_{10}) words; releasing the library will reclaim an additional /600 (384_{10}) words. Attempting to use WHAT or the symbolic library after it is released will cause a compile error.

(13) DEBUG n

This system statement is designed for the user with some knowledge of G-20 machine code and a general knowledge of the Algol-20 translator who wishes more specific information on the translation of a particular statement. This statement controls the printing of up to four columns of information after each ALGOL text card:

character scanned

postfix produced

code produced

internal variable equivalents

This information is printed one item per line as it is generated. An internal variable equivalent is printed out for each identifier declared. $n > 0$ turns on the printing, $n = 0$ turns it off.

CHAPTER 5

THE ALGOL LIBRARY

The primitive operators available to the ALGOL programmer include arithmetic operators such as +, *, and ↑ (exponentiation), and elementary mathematical functions such as SIN, EXP and ARCTAN. However, the programmer may need matrix inversion, numerical integration, least-squares curve-fitting, or calculation of eigenvalues and eigenvectors as basic operations in the solution of a particular programming problem. Since the last are somewhat less frequently used operations, they have not been provided as a part of the ALGOL language itself. Instead, these and other standard procedures are provided in a procedure library from which the programmer may call any library procedures which he needs in a particular program. Since the library is in no sense complete, existing procedures of general interest or the need for new procedures should be brought to the attention of the Computation Center staff.

There are two libraries in the ALGOL-20 system: the relocatable library and the symbolic library. The relocatable library contains for the most part those procedures which must be coded in machine language, such as DISC.READ and DISC.WRITE. The routines in this library are assembled once by the Center staff and placed in the library as relocatable binary machine instructions. From there they may be accessed by the user and loaded into any ALGOL program. This loading process is significantly faster than compiling a copy of the procedure into the user's program.

The symbolic library contains pieces of ALGOL text. This text will typically be a procedure declaration, but it need not; text for several procedures, a block, or an arbitrary sequence of ALGOL instructions may be filed as a single entry in the symbolic source language library. Procedures in this library are usually those which can be written more conveniently in ALGOL than in machine language; for example, SIM, the Simpson's Rule integration procedure. While a more efficient procedure might be written in relocatable machine language, there are at least two advantages to writing the procedure in ALGOL: First, the routine can be written and debugged in less time and with less effort by using ALGOL, and second, ALGOL text is more easily read. Anyone who is interested in

the detailed operation of the procedure may get an ALGOL listing of the procedure as it is compiled into his program. (See the discussion of print-control statements in Chapter 4.)

Since all the objects in the libraries are not procedures, the term "routine" is applied to an entity in either library. Thus, a routine may be a closed subroutine, a procedure, several procedures or an arbitrary sequence of ALGOL instructions. The descriptions of all routines in this chapter state whether the particular routine is in the symbolic or relocatable library.

Since there are different processes involved in processing symbolic source language and relocatable binary routines, there are different mechanisms for accessing symbolic and relocatable routines. Symbolic library routines are accessed by means of a SYSTEM statement. (See Chapter 4 for a complete discussion of SYSTEM statements.) The card image

SY LIBRARY <identifier>

will cause the text for the routine <identifier> to be compiled into the program <u>at that point</u>. The routine may then be used in the same way as any other routine which appears in the program.

Relocatable routines are accessed by a slightly more complex mechanism; they must be <u>declared</u> as <u>library</u> procedures in the head of the block in which they are used. The syntax of declarations is extended to include a library procedure declaration:

<library procedure declaration> ::= <u>library</u> <u>procedure</u> <identifier list>
| <u>library</u> <type> <u>procedure</u> <identifier list>

Thus the following are examples of library procedure declarations:

<u>library</u> <u>procedure</u> DISC.READ, DISC.WRITE,SLEW;
<u>library</u> <u>real</u> <u>procedure</u> ZILCH, SINH, GOSH;

These declarations have the same scope as any other ALGOL declarations; thus the name of a library procedure may be redeclared in other blocks to be any other ALGOL construct. Therefore, it may be necessary to declare the same library procedure in several different blocks. No matter how many times the procedure is declared only <u>one</u> copy of the routine will be added to the program.

As with any other library, the ALGOL library also has a librarian, which is used to update and edit the library. Using the librarian, the user may add his own routines to the library on a temporary basis. For a description of the librarian, see the ALIBN Manual.

The remainder of this chapter is a set of descriptions of the routines currently available in the relocatable and symbolic libraries; these are arranged within each library alphabetically according to the procedure name. Also, for completeness, writeups are included for "standard routines" - those routines which are built into the system and whose names are reserved identifiers. As these routines are automatically included in any program which calls on them, they do not have to be declared. In fact, any attempt to fetch them with an SY LIBRARY card or with a library procedure declaration will be detected as an error.

The reference for numerical method given for many of these routines is "1604 Routines". This refers to the book, Some Basic 1604 Mathematical Subroutines, Publication 061 of Control Data Corporation, Minneapolis, Minnesota. A copy of this book is available at the Computation Center for reference.

Standard Function

PROCEDURE SPECIFICATION

real procedure ARCTAN (X); value X; real X;

PURPOSE

ARCTAN finds the inverse tangent of X in radians in the range from $-\pi/2$ to $+\pi/2$. It operates correctly on any number given as input.

METHOD

Described on page G-1 of "1604 Routines".

TIMING and ACCURACY

The result is produced in 1.25 milliseconds.

The error is less than $1_{10}-11$.

Standard Function

PROCEDURE SPECIFICATION

<u>real</u> <u>procedure</u> COS (X); <u>value</u> X; <u>real</u> X;

PURPOSE

COS finds the cosine of X, where X is in radians and may be either positive or negative.

METHOD

COS uses the sine routine, using the identity

$$COS\ (X) = SIN\ (X + \pi/2).$$

ALARMS

RUN ERROR - SIN␣ $|X| > 2{,}097{,}152 = 2 \uparrow 21$

For values beyond this point the algorithm breaks down.

TIME and ACCURACY

The result is produced in 1.08 milliseconds. The relative error is about 5_{10}-11.

ALGOL Symbolic Library

PROCEDURE SPECIFICATION

```
PROCEDURE CURFIT  (K, A, B, M, X, Y, W, N, ALPHA, BETA, S, SGMSQ, XO, GAMMA, C, Z,
                   R, ORTH, POW, ERROR);
VALUE K, M, N, XO, GAMMA, R; INTEGER K, M, N. R;
ARRAY A, B, X, Y, W, ALPHA, BETA, S, SGMSQ, C, Z;
REAL XO, GAMMA; BOOLEAN ORTH, POW; LABEL ERROR;
```

REFERENCES

1. Algorithm 74, Comm. ACM, January 1962
2. Peck, J.E.L. Polynomial Curve Fitting with Constraints, SIAM Review, April 1962, pp. 135-141

PURPOSE

CURFIT finds the polynomial of degree N which passes through the K points (A[1], B[1]),...,(A[K], B[K]) and fits the M points (X[1], Y[1]),...,(X[M], Y[M]) in the least squares sense, where W[I] is the weight attached to the point (X[I], Y[I]), I = 1,...,M. The coefficients of this polynomial are stored in the array C; C[J] is the coefficient of X^J for J = 0,...,N. The sum of the squares of the deviations is computed for each polynomial of degree greater than K-1 but not greater than N, and is stored in the array SGMSQ, i.e., if $F_L(X)$ is the least squares polynomial of degree L, then

$$\mathrm{SGMSQ}[L] = \sum_{I=1}^{M} (Y[I] - F_L(X[I]))^2 \quad , L = K,\ldots,N .$$

CURFIT will also evaluate the polynomial which it has determined, if the user desires, for the set of values of the independent variable which it finds in the array Z. The options available are explained under USAGE.

RESTRICTIONS

(Note: In all that follows, upper case letters refer to formal parameters, while lower case letters denote the corresponding actual parameters.)

1. In the calling program, the arrays used by CURFIT must be declared to include the subscript bounds as shown below.
   (a) Input Arrays:
       <u>real</u> <u>array</u> a, b[1:k], x, y, w[1:m], z[1:r];
   (b) Output Arrays:
       <u>real</u> <u>array</u> alpha, beta [0:n-1], s, c[0:n] sgmsq[k:n];

2. The values of the actual parameters must satisfy the following conditions:
   (a) $1 \le k \le n < m + k$
   (b) $m \ge 1$
   (c) gamma $\neq$ o

3. The user is reminded of the ALGOL-20 restriction regarding labels used as actual parameters in procedure calls. See page AL.2.14 of the ALGOL-20 Manual for details. The conditions which will cause the procedure to transfer control to the statement which has the label error are:
   (a) One or more of restrictions 2 (a), (b) has been violated.
   (b) Division by zero was about to be attempted during a calculation. The usual cause of this is improper or inconsistent data in the input arrays x and y.

4. The arrays a and x must not contain an element in common, or the effect of the procedure is undefined.

5. CURFIT destroys the contents of the arrays a, x, y and w. Consequently, provision must be made to save these data (if desired) before CURFIT is called.

METHOD

CURFIT uses the method of orthogonal polynomials, which can be defined recursively by

$$p_{-1}(t) = 0 \quad , \quad p_0(t) = 1 \; ,$$

$$p_{i+1}(t) = (t - \alpha_i)\, p_i(t) - \beta_i p_{i-1}(t) \; , \; i = 0,\ldots,n-1. \qquad (1)$$

The coefficients α_i , β_i and s_i are determined such that the nth degree polynomial

$$F_n(t) = s_0 p_0(t) + s_1 p_1(t) + \ldots + s_n p_n(t) \qquad (2)$$

minimizes the quantity

$$\delta_n^2 = \sum_{r=1}^{m} w_r \left[y_r - F_n(t_r) \right]^2$$

The coefficients s_i can now be used to compute the coefficients of the standard polynomail representation of F_n . Suppose that $V_{i,r}$ is the coefficient of t^r in $p_i(t)$, i.e.,

$$p_i(t) = V_{i,0} + V_{i,1} t + \ldots + V_{i,i} t^i \; .$$

Then from equation (1), with $V_{0,0} = 0$, $V_{i,-1} = 0$ for all i, and $V_{i,r} = 0$ for $r > i$, we have

$$V_{i+1,r} = V_{i,r-1} - \alpha_i P_{i,r} - \beta_i V_{i-1,r}$$

for $0 \leq i < n$ and $0 \leq r \leq n$. Hence by equation (2),

$$c_r = \sum_{i=r}^{n} s_i V_{i,r} \quad , \quad r = 0,\ldots,n,$$

and we have

$$F_n(t) = c_0 + c_1 t + \ldots + c_n t^n$$

USAGE

1. Round off errors are reduced if all the abscissas lie in the interval [-2,2] (see Reference 2). Consequently, a change of scale is introduced by the procedure, using the transformation

$$X'[I] \leftarrow (X[I] - XO) / GAMMA ,$$

where XO and GAMMA must be supplied by the user in accordance with the size of the data. The appropriate values of these parameters can be determined from the equations

$$x0 = 1/2 [\max(x[I], a[I]) + \min(x[I], a[I])]$$

$$gamma = 1/4 [\max(x[I], a[I]) - \min(x[I], a[I])].$$

2. The coefficients α_i, β_i, s_i, and c_i, as well as the sums of the squared deviations, are accessible to the user as the contents of the arrays alpha, beta, c, s, and sgmsq, respectively.

3. In addition, there are two built-in print options:

(a) If orth is true then the value of the least squares polynomial $F_n(t)$ is computed using equation (2) for each element of the array Z. A message to this effect is printed, followed by r rows of output, each consisting of the number (subscript) J of the array element, the value z[J] of the array element, and the value $F_n(z[J])$ of the polynomial.

(b) If pow is true, then the value of the least squares polynomial $F_n(t)$ is computed using equation (3). The format of the output is the same as part (a).

In some cases method (a) will yield more accuracy then method (b). However, the corresponding values will usually agree to four or five significant figures.

TIME AND ACCURACY

With all internal printing turned off (orth and pow both false), CURFIT determined an eighth degree polynomial, passing through two points and approximating eight other points, in approximately one second. The sum of the squares of the deviations at the eight approximated points was approximately 10% of the difference between the maximum and minimum ordinates of the curve.

ALGOL Symbolic Library

PROCEDURE SPECIFICATION

```
PROCEDURE ELIPS (M1, K, E, TOL, ALARM);
VALUE M1, TOL; REAL M1, K, E, TOL; LABEL ALARM;
```

REFERENCES

1. Algorithm 165, Comm. ACM, April 1963
2. M. Abramvotiz and I. A. Stegun, Handbook of Mathematical Functions, National Bureau of Standards, 1964, p. 598.

PURPOSE

Given a value of the (complementary) parameter M1, this procedure computes the numerical values of K and E, the complete elliptic integrals of the first and second kinds, which are defined by

$$\text{First kind:} \quad K(M1) = \int_0^{\pi/2} \frac{dx}{\sqrt{1 - (1-M1)\sin^2 x}}$$

$$\text{Second kind:} \quad E(M1) = \int_0^{\pi/2} \sqrt{1 - (1 - M1)\sin^2 x}\; dx$$

RESTRICTIONS

1. The user is reminded of the ALGOL-20 restriction regarding labels used as actual parameters in procedure calls. See page AL.2.14 of the ALGOL-20 Manual for details.

2. If $M1 \leq 0$ or $M1 > 1$ the elliptic integrals are undefined, and the procedure transfers control to the statement whose label is the actual parameter corresponding to ALARM.

3. The actual parameter corresponding to TOL determines the accuracy (and also the execution time) of the procedure. The size of this parameter is limited by the relative accuracy of the built-in ALGOL square root routine, and must not be less than 10^{-12}

METHOD

The arithmetic-geometric mean process is used (see reference 2). Starting with the triple $(a_0, b_0, c_0) = (1, \sqrt{ml}, 1 - ml)$, new values are computed using the iterative scheme

$$a_i = 1/2\ (a_{i-1} + b_{i-1}) \qquad b_i = \sqrt{a_{i-1}\ b_{i-1}} \qquad c_i = 1/2\ (a_{i-1} - b_{i-1}).$$

During the calculation, the quantity S_i, defined by

$$S_i = \sum_{j=0}^{i} 2^j c_j^2 ,$$

is accumulated. The process stops when the two conditions

$$\frac{|c_i|}{a_i} < TOL \quad \text{and} \quad \frac{2^i c^2_i}{S_{i-1}} \leq TOL$$

are both met. The desired elliptic integrals are then found using the relations

$$K = \frac{\pi}{2a_i} \qquad E = K(1 - 1/2\ S_i).$$

TIME AND ACCURACY

Using a value of TOL = 10^{-4}, nine to ten significant decimal digits were obtained for both elliptic integrals in approximately ten milliseconds of computation time.

Standard Function

PROCEDURE SPECIFICATION

real procedure EXP (X); value X; real X;

PURPOSE

EXP computes the exponential function

E↑X where E = 2.71828182845...

METHOD

A continued fraction approximation is used.

For further details see page AA-42 of "1604 routines".

RANGE

For X < -161, the result is given as zero.

It is an error if X >160.116998, since in that case EXP(X) will exceed the largest number representable in the 6-20.

ALARMS

RUN ERROR - EXP␣ if X > 160.116998.

TIMING and ACCURACY

The result is produced in about 1 to 2 milliseconds.

The error is less than $1_{10}-10$.

ALGOL Symbolic Library

PROCEDURE SPECIFICATION

```
PROCEDURE FREQ (N, A, B, IUL, K, X, KA);
INTEGER N, IUL;  INTEGER ARRAY KA;
REAL A, B, K; REAL ARRAY X;
```

REFERENCE

Algorithm 212, Comm. ACM, October 1963

PURPOSE

FREQ determines the frequency distribution of N real variables, the elements of the vector X, over the interval [A, B]. Each element of X is assigned to one of K equal, half-open subintervals of [A, B], and the frequency of X in the Jth subinterval is stored in KA[J].

RESTRICTIONS

1. A is assumed to be the left end-point of the interval, and hence must be less than B.

2. The elements of the vector X must satisfy the inequalities $A \leq \min (X[I]) < \max (X[I]) < B$ for $I = 1, \ldots, N$.

3. The number N of variables being classified must be less than or equal to the order of the array X.

4. Upon entry, the array KA is assumed identically zero. In the calling program, the array declaration for the actual parameter ka corresponding to the formal parameter KA should be: <u>integer</u> <u>array</u> ka [1 : k]; where k is the actual parameter corresponding to K.

METHOD

The interval [A, B] is transformed into the interval [0, K] with unit subintervals and the elements of the array X classified there by

$$Y[I] = (X[I] - A)/((B - A)/K) \text{ for } I = 1,\ldots,N.$$

USAGE

The user has the option of having the subintervals closed at either the upper or lower end, and must specify his choice according to the following scheme: If FREQ is called with IUL = 0, then the <u>lower</u> end-point is included and the upper end-point is omitted from each subinterval (except, of course, the Kth). If IUL $\neq$ 0, then the <u>upper</u> end-point is included and the lower omitted, in each subinterval except the first.

TIME

FREQ classifies approximately one thousand numbers per second.

ALGOL Symbolic Library

PROCEDURE SPECIFICATION

PROCEDURE GJR(A, N, EPS);

VALUE N, EPS; ARRAY A; INTEGER N; REAL EPS;

REFERENCES

1. Certification of Algorithm 120, Comm. ACM, Jan. 1963
2. H.R. Schwarz, An Introduction to ALGOL, Comm. ACM, February 1962, p. 94

PURPOSE

GJR computes the inverse of the N by N matrix A and stores the resulting inverse in A.

RESTRICTIONS

1. The actual parameter a which corresponds to the matrix A must be declared in the calling program as: REAL ARRAY a [1 : n, 1 : n], where n is the actual parameter corresponding to N.

2. An exit, labeled SINGULAR, must be supplied in the main program. If any pivot element of the given matrix is less than EPS in absolute value, control will be transferred to the statement which has the label SINGULAR.

3. The parameter EPS is impossible to describe in absolute terms; the correct value to use depends upon the matrix being inverted and also on the precision of the computer. If EPS is too large, GJR will not be able to invert the matrix and will exit to the label SINGULAR, in which case the contents of the array corresponding to A will be meaningless. If EPS is too small and the given matrix is ill-conditioned (close to being singular), the results may be meaningless anyway due to round-off errors in division.

The following scheme is suggested: Use an EPS in the range 10^{-5} to 10^{-7}. If GJR inverts the matrix and there is any doubt that the inverse is correct, multiply the original matrix by the calculated inverse and compare the result with the unit matrix to determine the accuracy of the inverse. If GJR does not invert the matrix, decrease EPS (e.g., divide by 10) and repeat until an inverse is obtained; check this inverse by the above method.

4. The order of the matrix to be inverted must not exceed seventy.

METHOD

The Gauss-Jordan direct elimination method, preceded by a pivotal search, is used. See K.S. Kunz, Numerical Analysis, McGraw-Hill, 1957, pp. 220-22, 234 and reference 2 for details.

USAGE

The user is warned again that GJR replaces the given matrix by its inverse. Consequently, if e.g., the matrix to be inverted arises as a result of a computation and is to be saved, provision must be made in the main program to store this array elsewhere or else print it out before GJR is called.

TIME and ACCURACY

GJR requires approximately $6.5N^3 * 10^{-4}$ seconds of execution time, where N is the order of the matrix being inverted. Typical accuracy is eight significant decimal digits.

ALGOL Relocatable Library

PROCEDURE SPECIFICATION

```
procedure GOOF.STAR (PROCEDURE.NAME, ERROR.CODE);
    value ERROR.CODE;
    procedure PROCEDURE.NAME; string ERROR.CODE;
```

PURPOSE

GOOF.STAR calls the ALGOL run-error routine which prints the diagnostic error message described in Chapter 6b. This routine may be of particular use to the programmer writing routines for the ALGOL Symbolic Library who wishes to use the ALGOL run-error machinery.

RESTRICTIONS

The actual parameter PROCEDURE.NAME must not be a function designator; that is, it must have been declared as a procedure, not as a real procedure, Boolean procedure, etc.

METHOD

A call of GOOF.STAR causes the ALGOL error routine to print the diagnostic message

RUN ERROR - eeee

where 'eeee' is the actual parameter corresponding to ERROR.CODE. The name of the procedure must be given as PROCEDURE.NAME so that the error routine can find the entry to the erring routine and thus print the commands in the user's program which (presumably) caused the error. The error routine will execute a HALT, unless an error-recovery switch has been set by RUN.ERROR.

USAGE

Suppose a procedure PIP is used, one of whose parameters is a non-negative number, DELTA. The statement within PIP

```
if DELTA < 0 then GOOF.STAR (PIP, 'PIP1');
```

will check the validity of the given DELTA and, if there is an error, will call GOOF.STAR to print

RUN ERROR - PIP1

and the usual diagnostic information. The user may then look in the ALARMS section of the description of PIP and discover that the PIP1 error resulted because DELTA was negative. The user may use the procedure RUN.ERROR to recover from error conditions which invoked calls of GOOF.STAR.

ALARMS

A run-error GOOF will result if the parameter procedure.name is the name of a function designator.

ALGOL Relocatable Library

PROCEDURE SPECIFICATION

procedure GO.SEG (I);

value I; *integer* I;

PURPOSE

GO.SEG positions the segment tape to segment i. The use of this routine can substantially increase the running speed of a segmented program.

METHOD

In the usual case, some computation will be carried out before a new segment is to be loaded. The statement

GO.SEG (i);

will initiate a slew to segment i. Computation will resume until the next call for LINK. At that time, computation will cease until the tape is positioned and the segment is loaded. Thus, it is possible to overlap time spent slewing to a segment with useful computation. Since it takes about

$$(600 + 500 * |N-M-1|) \text{ ms}$$

to slew a tape from segment M to segment N, it is worthwhile to overlap as much slew time as possible.

ALGOL Relocatable Library

PROCEDURE SPECIFICATION

```
procedure  LINK (I);
     value I; integer I;
```

PURPOSE

LINK loads segment i into core and enters it at the first begin.

METHOD

LINK loads segment i from bulk storage, reading only as many words as there are in the compiled segment. If the calling program and the called segment have identical declarations of own scalars and own arrays, none of these own variables will be disturbed.

Executing LINK causes all non-own variables to be made undefined. In addition, certain internal variables are reset:

(1) The READ, PUNCH, and PRINT buffers, which may have been set by BUFFER.SET, are reset to the standard buffers supplied by ALGOL.

(2) All error-recovery switches created by RUN-ERROR are cleared.

Finally, control is transferred to the outermost begin of the newly loaded segment; that is, the first statement executed is always the first statement of the segment.

ALARMS

A run error LINK will occur if

(1) $i < 1$, or if

(2) segment i was not previously dumped as an ALGOL segment.

TIMING

Each call of LINK takes approximately

$$(600 + 500 * |N-M-1|)\text{ms}$$

where M is the segment at which the tape is currently positioned and N is the segment to be loaded. Running time can frequently be substantially reduced by use of the procedure GO.SEG.

Standard Function

PROCEDURE SPECIFICATION

<u>real</u> <u>procedure</u> LN (X); <u>value</u> X; <u>real</u> X;

PURPOSE

LN computes the natural logarithm of X.

METHOD

A Chebychev polynomial approximation is used.

For further details see page AA-17 of "1604 Routines".

ALARMS

RUN ERROR - LN␣␣ if $X < 0$.

TIMING and ACCURACY

The result is produced in 1.75 milliseconds.

The error is less than $5_{10}-11$.

ALGOL Symbolic Library

PROCEDURE SPECIFICATION

```
PROCEDURE MULLER (P1, P2, P3, MXM, NRTS, EP1, EP2, SW1, SW2, SW3, SWR, RRT, IRT,
                  FUNCTION);
VALUE P1, P2, P3, MXM, NRTS, EP1, EP2, SW1, SW2, SW3, SWR; INTEGER MXM, NRTS;
BOOLEAN SW1, SW2, SW3, SWR;  REAL P1, P2, P3, EP1, EP2;  ARRAY RRT, IRT;
PROCEDURE FUNCTION;
```

REFERENCES

1. Algorithm 196, Comm ACM, August 1963.
2. Muller, D.E., A Method for Solving Algebraic Equations Using an Automatic Computer, MTAC, vol. 10 (1956), pp. 208-215.
3. Frank, W.L., Finding Zeros of Arbitrary Functions, JACM, Vol. 5, No. 2 (1958) pp. 154-160.

PURPOSE

MULLER solves a general equation of the form f(z) = 0, where f(z) is analytic in a neighborhood of the roots. The value of parameter NRTS is the number of solutions to be determined. Both real and complex roots are found; the Ith root determined is stored by placing the real part in RRT[I] and the imaginary part in IRT[I] for I = 1,...,NRTS. Various print options and search techniques are controlled by the Boolean parameters SW1, SW2, SW3, and SWR, as explained below. No prior knowledge of the location of the roots is required. Multiple roots are also obtainable, although with less accuracy than for isolated roots.

RESTRICTIONS

(Lower case letters denote actual parameters, while upper case letters refer to formal parameters.)

1. The arrays rrt and irt must be declared in the calling program to include the subscript bounds shown below:

```
real array rrt, irt[1:nrts]
```

2. The procedure function must be declared in the program before MULLER is called. It must be able to supply the real and imaginary parts of the value of f(z) for any desired value of z = x + iy. The declartion of this procedure must be as follows:

```
procedure function (RE.Z, IM.Z, RE.F, IM.F);
value  RE.Z, IM.Z; real RE.Z, IM.Z, RE.F, IM.F;
     begin  ...
        (procedure body)
                ...
     end;
```

In the above declaration, RE.Z and IM.Z are the real and imaginary parts of the independent variable, while RE.F and IM.F are the real and imaginary parts of the function

evaluated at the point (RE.Z, IM.Z).

Observe that since FUNCTION is a parameter of MULLER, several equations may be solved in the same program by declaring the appropriate function generator for each equation with a different identifier. If this technique is employed, the user must either: (a) print out the contents of rrt and irt after each call of MULLER; or (b) supply a different pair of arrays corresponding to RRT and IRT in each call of MULLER.

3. If the equation to be solved is known to have only real roots, swr must have the value <u>true</u> when MULLER is called.

4. p1, p2, and p3 are the real parts of three arbitrary starting points. If the equation to be solved is suspected to have multiple roots, the choice of p1, p2, and p3 should be such that no one of them is exactly equal to a multiple root. If this restriction is not met, MULLER will not discover that the function has multiple zeros at this particular root.

5. If the equation to be solved is a polynomial, sw3 should be set <u>true</u>. This causes the procedure to accept the conjugate of each root found as a root, when applicable (i.e., when the root is not real).

6. If the function has singularities, p1, p2, and p3 should be chosen such that none of them is a singular point.

7. ep1 and ep2 are parameters which specify convergence criteria, while the value of mxm dictates the maximum number of iterations to be made in locating any one root. If neither of the convergence criteria has been satisfied after mxm iterations, the most recent iterant is accepted as a root and MULLER will proceed to find the next root (or terminate if nrts roots have been found). The user will receive no warning that this event has occurred unless sw2 has the value <u>true</u>. When sw2 is <u>true</u>, each root is printed as it is found; thus if convergence does not occur, a message to that effect will be printed along with the value of the last iterant and the value of the function at that point.

<u>METHOD</u>

The algorithm used by MULLER is as follows: given an arbitrary function $f(z) = 0$ which can be evaluated for any value of z and is analytic in a neighborhood of its roots, select three arbitrary starting values z_1, z_2, z_3. Find the second degree (Lagrange) polynomial which passes through the three points $(z_1, f(z_1))$, $(z_2, f(z_2))$ and $(z_3, f(z_3))$. Choose z_4 to be the root of this polynomial which lies closer to z_3. Then drop z_1 and repeat the process with z_2, z_3, and z_4, etc.

The process stops when any of the following three conditions is met:

(1)
$$\frac{z_{i+1} - z_i}{z_{i+1}} < ep1$$

where z_i is the i-th iterant.

(2) $|f(z_i)| < ep2$ <u>and</u> $|f_r(z_i)| < ep2$ where $f(z_i)$ is the value of the function at the point z_i, and $f_r(z_i)$ is the value of the "modified function" evaluated at z_i.

(3) The number of iterations is equal to mxm.

The "modified function" is defined as

$$f_r(z) = \frac{f(z)}{\prod_{j=1}^{r-1} (z - z_j)} \qquad \text{for } r = 2, 3, \ldots \qquad (1)$$

where z_j is the j-th root found. If f(z) is not a polynomial, it is not possible to divide out the roots as they are found in order to reduce the degree of the polynomial and hence the amount of computation. This device is valid, however, if the division is performed only for those numbers z actually entering the algorithm, i.e., the points z_i which are generated during each iteration. Thus, having found r-1 roots, the r-th root is determined by applying the MULLER algorithm to the equation $f_r(z) = 0$, where $f_r(z)$ is defined by equation (1). This technique should succeed only for functions f(z) which have no multiple roots, since $f_r(z)$ as defined above is indeterminate when $z \to z_r^*$, where z_r^* is a root previously found. However, due to the lower accuracy with which multiple roots are determined, they in effect behave like clustered roots and hence the device has not yet failed.

USAGE

The region near the starting points $(p_1, 0)$, $(p_2, 0)$, $(p_3, 0)$ is examined first for roots; successive roots are then found (usually) in order of increasing magnitude. Thus it is suggested that starting values reasonably close to the origin be used, unless it is known that the equation has no roots in that vicinity. In the absence of any knowledge about the solutions of the equations, it has been found (reference 2) that the starting value pl = -1, p2 = 0, p3 = 1 lead to good performance of the algorithm.

The role of the Boolean parameters will be reviewed here briefly.

(1) If swl is true, then each iterant of each root is printed, along with the corresponding values of the function and modified function.
(2) If sw2 is true, then each root is printed as it is found, along with the corresponding values of the function and modified function.
(3) If sw3 is true, then the complex conjugate of the root just found is, when the root is not real, admitted as a root. It is computed directly from the value of the previous root, rather than carrying out the iterative process for its determination.
(4) If swr is true, then the imaginary part of each iterant is forced to be zero throughout the iteration, and hence only real roots will be found.

TIME AND ACCURACY

Since it is impossible to adequately describe in general terms the required execution time and obtainable accuracy for this procedure, several examples are given which indicate its performance. All results were correct to ten significant decimal digits.

(1) Using the call

MULLER (1, -1, .5, 50, 10, ₁₀-10, ₁₀ -10, false, false, true, false, X, Y, FUNCTION);

all ten roots of the equation $z^{10} - 1 = 0$ were found in six seconds.

(2) The five roots closest to the origin (which all lie on the negative real axis) of the equation

$$\sin x = \frac{x + 1}{x - 1}$$

were determined in 44 seconds with the call

MULLER (-.4, -5.5, -3.8, 20, 5, $_{10}$-10, $_{10}$-10, <u>true</u>, <u>true</u>, <u>false</u>, <u>true</u>,
X, Y, FUNCTION);

(3) The single real root of the equation

$$xe^x = 2$$

was located in seven seconds with the call

MULLER (-1, 0, 1, 20, 1, $_{10}$-10, $_{10}$-10, <u>true</u>, <u>true</u>, <u>false</u>, <u>true</u>,
X, Y, FUNCTION);

(4) Fifty-three seconds were required to determine the eight roots of the equation

$$z^8 + z^7 + 3z^6 + 2z^5 + 3z^4 - z^3 - 2z^2 - 2z + 1$$

using the call

MULLER (-1, 1, 0, 20, 8, $_{10}$-10, $_{10}$-10, <u>true</u>, <u>true</u>, <u>true</u>, <u>false</u>, X, Y,
FUNCTION);

ALGOL Symbolic Library

PROCEDURE SPECIFICATION

```
REAL PROCEDURE NEVILLE (W, X, Y, N, P, ALARM, EXTRAPOLATE);
VALUE W, N, P;  ARRAY X, Y;  REAL W;  INTEGER N, P;
LABEL ALARM;  BOOLEAN EXTRAPOLATE;
```

REFERENCES

1. W. E. Milne, Numerical Calculus, p. 73
2. J. Todd, Survey of Numerical Analysis, McGraw-Hill, 1962, pp. 39-42

PURPOSE

Given N abscissas X[0],...,X[N-1] and N corresponding functional values Y[0],...,Y[N-1] which are related by Y[I] = f(X[I]), I = 0,...,N-1 , this procedure performs a P-point interpolation to find the approximate functional value corresponding to the input abscissa W, and stores the result in NEVILLE. If the value of W is outside the range of the table (i.e., W < X[0] or W > X[N-1]), and EXTRAPOLATE has been set false then the procedure transfers control to the statement whose label is the actual parameter corresponding to ALARM; if EXTRAPOLATE is true, the P points closest to the appropriate end of the table are used to compute the approximate value of f(W) by extrapolation.

RESTRICTIONS

(Note: Formal parameters are denoted by upper case letters, and actual parameters by lower case letters.)

1. Alarm must be declared in the calling program as a label. See page AL.2.14 of the ALGOL-20 Manual.

2. The procedure assumes that there are n points in the table having subscripts 0 through n - 1. Consequently, when the arrays x and y are declared in the calling program, their subscript bounds must include those shown in the declaration

real array x, y[0:n-1];

3. The abscissas must be monotone increasing, i.e., x[I] > x[I-1] for I = 1,...n - 1, but they need not be evenly spaced.

4. The number of points used in the interpolation cannot exceed the number of entries in the table, i.e., n ≥ p .

METHOD

A variation due to Neville of Aitken's iterative interpolation scheme is used. This technique is equivalent to computing the value (at the point of w) of the (p-1)-st degree Lagrange polynomial which passes through the p points closest to w.

USAGE

If either of restrictions 3 or 4 are violated the effect of the procedure is undefined. In the event that x[I] = x[J] for $I \neq J$, a division by zero will occur and the execution of the program will be terminated with an EXPONENT OVERFLOW message. Otherwise NEVILLE will attempt to carry out the calculations, but will give meaningless results. Consequently, the user should take special care to ensure that the restrictions are met.

TIME AND ACCURACY

The approximate execution time is

$$0.6p^2 + 0.6p + 0.05n + 0.6$$

milliseconds, where p and n are the actual parameters which have been defined above. The accuracy depends upon the behavior of the tabulated function and the number of points used in the interpolation. Seven to eight significant figures were obtained with $n = 20$ and $p = 5$ for the functions SIN(X), COS(X) and EXP(X) over the interval [0,1].

ALGOL Symbolic Library

PROCEDURE SPECIFICATION

PROCEDURE NORMRAN (MU, SIGSQR, NR, M, SV);

REAL ARRAY NR; INTEGER M; REAL MU, SIGSQR, SV;

REFERENCE

R. W. Hamming, Num. Meth. for Science and Engineering, McGraw-Hill, 1962, p. 389

PURPOSE

NORMRAN generates a sequence of M pseudo-random numbers, normally distributed with mean MU and variance SIGSQR, and stores the sequence in the vector NR.

RESTRICTIONS

(1) The starting value SV must be an odd positive eleven digit number in integer form (no decimal point). SV is used to supply the subprocedure RANDOM which generates uniformly distributed pseudo-random numbers.

(2) The actual parameter which is substituted for SIGSQR in the call of NORMRAN must have a positive value.

(3) The desired amount of numbers M must be less than or equal to the order of the array which is the actual parameter corresponding to NR.

(4) RANDOM is a procedure local to NORMRAN.

METHOD

NORMRAN makes use of RANDOM and the central limit theorem of probability in the following way: If X[J] is a uniformly distributed random number with variance V, then the sequence of numbers

$$NR[I] = \mu + \sqrt{\sigma^2}\ \left(\sum_{J=1}^{N} X[J]\right)\ \sqrt{V/N}\ ,\ I = 1, \ldots, M.$$

very nearly approximates a sequence of normally distributed random numbers with mean μ and variance σ^2 being good for $N \geq 10$. Procedure NORMRAN adds the X[J] in blocks of twelve. Since $-1 \leq X[J] \leq 1$ for all J, the above formula reduces to

$$NR[I] = \mu + (\sqrt{\sigma^2}/2) \sum_{J=1}^{12} X[J],\ I = 1, \ldots, M.$$

TIME

NORMRAN generates and stores one hundred numbers in approximately 3 seconds.

AL.5.NORMRAND.2

ALGOL Symbolic Library

PROCEDURE SPECIFICATION

REAL PROCEDURE RANDOM (A, B, XO);

VALUE A, B, XO; REAL A, B, XO;

REFERENCES

1. Algorithm 133, Comm. ACM., Nov. 1962
2. R. W. Hamming, Num. Meth. for Scien. and Engr., McGraw-Hill, 1962, p. 384.

PURPOSE

RANDOM generates the next member of a sequence of pseudo-random numbers from a uniform (rectangular) distribution on the interval (A, B).

RESTRICTIONS

1. The first time RANDOM is called, the starting value XO should be an odd, positive, eleven digit integer; on subsequent calls in the same program, use XO=0. The procedure declares an *own* variable which saves the current value of RANDOM for use as a starting value on successive calls.

2. A is assumed to be the left end point of the interval of the distribution, so that in most applications A should be less than B.

METHOD

The method of congruences is used for the generation:

$$\text{RANDOM}_{i+1} \leftarrow 5 * \text{RANDOM}_i \pmod{2^{35}}.$$

If RANDOM is called repeatedly, this results in a sequence with a period of 2^{33}.

USAGE

As an example of the use of RANDOM, the following short ALGOL program generates 5000 pseudo-random numbers uniformly distributed on the interval (0,1) and stores the numbers in the vector DATA:

```
AL      begin real array DATA [1:5000];
        integer I ;
SY      LIBRARY RANDOM
        DATA [1] ← RANDOM (0,1, 13543288579) ;
        for I ← 2 step 1 until 5000 do
                DATA [I] ← RANDOM (0, 1, 0);
        end
```

TIME

Approximately 2.2 seconds are required for the generation of one thousand pseudo-random numbers using this procedure.

ALGOL Relocatable Library

PROCEDURE SPECIFICATION

```
procedure RUN.ERROR(L1, ERROR.TYPE, ERROR.PRINTING);
     value ERROR.TYPE, ERROR.PRINTING;
     label L1; string ERROR.TYPE; Boolean ERROR.PRINTING;
or
procedure RUN.ERROR(L1, ERROR.TYPE);
     value ERROR.TYPE; label L1; string ERROR.TYPE;
```

PURPOSE

There are certain programming problems in which the programmer is able to predict that error conditions, as detected by ALGOL at runtime, may well occur in some data sets. It is, however, frequently as much trouble to check a data set for errors as to process the data set. Thus bad or missing data may lead to subroutine errors, exponent overflows, or address-opcode faults. The use of RUN.ERROR allows the programmer to recover control in such situations, print diagnostic information, correct for the error, and continue processing.

METHOD

Error conditions detected by the G-21 hardware or by a subroutine during the execution of ALGOL program ordinarily result in the message

RUN ERROR - eeee

and the termination of the run; the string 'eeee' specifies the type of error detected. In the case of errors detected by library procedures, 'eeee' is specified in the ALARMS section of the procedure description; for other runtime error codes and details concerning the diagnostic messages, see Chapter 6b.

To recover from such an error condition, the programmer must have previously specified which error types are to result in recovery and where in his program control is to be transferred for each of these error conditions. The first two parameters in the call of RUN.ERROR give this information: The value of ERROR.TYPE should be the string 'eeee' to specify recovery from error condition eeee. If error condition eeee arises, control is transferred to the label L1, subject to the rules

discussed below. The parameter L1 usually should be a simple label and not the more general form of designational expression; for a discussion of this see restriction 2.

The parameter ERROR.PRINTING, if included in the call, controls the printing of the usual diagnostic message for all error conditions. (See Chapter 6b for a description of this diagnostic message.) If ERROR.PRINTING = false, the diagnostics will not be printed; if ERROR.PRINTING = true, diagnostics will be printed. ERROR.PRINTING is used to set the run-time error printing mode switch (— 42) which is interrogated by the error recovery machinery each time it is called. If ERROR.PRINTING = false, no diagnostics will be printed for any routine for which error recovery is set up. This switch is only changed when a call of RUN.ERROR is executed which has three parameters. Thus if the programmer wishes to get only his own diagnostic printing, he may use the first form of call for RUN.ERROR, with ERROR.PRINTING = false. If an error occurs for which a recovery switch was not set, the usual diagnostic message will be printed. ALGOL initially assumes ERROR.PRINTING = true. (See also restriction 4.)

Scope of Error Recovery

For each call, RUN.ERROR creates a triple of the form

(L1, ERROR.TYPE, block,level),

hereafter called an error-recovery switch. The scope of these switches is determined by the block structure of ALGOL.

The use of any error-recovery switch will clear the switch. Thus if the programmer wishes to re-enable error recovery after having recovered from an error, he must reset the switch by again calling RUN.ERROR. The use of one error-recovery switch will not affect the status of any others which have been set.

Blocks, procedure declarations, and procedure calls may be nested arbitrarily in ALGOL. Since it may be convenient to have different recovery procedures for the same error condition when it occurs in different blocks, RUN.ERROR keeps the error-recovery switches in a stack. Thus error-recovery switches created in a given block will not destroy those set in an outer block, but will merely "push them down" in the stack. A newly created

switch will be effective for the block in which it is created and for all blocks and procedure calls nested in that block.

Leaving a block at dynamic level N will pop up the stack, deleting all error-recovery switches with block.level = N. Calling RUN.ERROR twice in the same block for the same ERROR.TYPE with actual parameters L1 and L2 will replace the triple (L1, ERROR.TYPE, block.level) by the triple (L2, ERROR.TYPE, block.level).

Monitor-Detected Errors

RUN.ERROR called with ERROR.TYPE = 'TIMR' will permit recovery from TIME LIMIT EXCEEDED, PAGE LIMIT EXCEEDED, OPERATOR TERMINATED, and MACHINE ERROR. The user's program will regain control and will be allowed an extra 30 seconds of running time. Repeated calls with the parameter 'TIMR' will not give additional intervals of 30 seconds.

One Procedure with Several Error Exits

While some complex procedures have several error exits, the programmer may wish to use the same recovery technique in more than one case. Error codes for such procedures are typically of the form 'ABCn', where n is a non-zero digit. If an error 'ABCn' occurs, RUN.ERROR checks for an error-recovery switch of the form ('ABCn', ,); if there is none, RUN.ERROR will then look for a switch of the form ('ABC0' , ,). (Note that the fourth character is zero, not the letter "O".) Thus the programmer may handle certain error conditions by special means and process any other errors by a single general mechanism.

Example: The error codes for DISC.WRITE are 'RWR1' ,..., 'RWR5'. Suppose the following calls of RUN.ERROR are executed:

```
RUN.ERROR (L1, 'RWR5');
RUN.ERROR (L3, 'RWR0');
```

If an error RWR5 occurs in the block, control will be transferred to L1. Any other DISCWRITE error will cause control to be transferred to L3.

USAGE

The following block of ALGOL code could be used to evaluate the function

$$y = \frac{x^3 + 1}{x}$$

For various values of x, printing out 'y IS INFINITE AT ...' whenever an exponent overflow occurred:

```
begin label L1; real x, y; library procedure RUN.ERROR;
comment: Note that since the first occurrence of L1 is
as a parameter to a proecure, we must declare it as a
label. See page 2.14 of this report;
     RUN.ERROR(L1, 'EXPO', false);
     for x:= -4 step 1 until 4 do
          begin y:= (x↑3 + 1) / x;
          NAME(x, y); PRINT (<-1D.1Z, 10B, -4D.4Z, E>);
          go to end.of.loop;
          L1:  NAME (x); PRINT(<'y IS INFINITE AT ', -1D.1Z, E>);
          RUN.ERROR(L1, 'EXPO');
          end.of.loop:  end
end
```

RESTRICTIONS

1. Not more than 20 distinct error-recovery switches are allowed at any one time.

2. While ERROR.TYPE and ERROR.PRINTING are called by value and thus are evaluated when RUN.ERROR is called, the label L1 is not evaluated until the error condition ERROR.TYPE actually occurs. If L1 is actually a general designational expression, it will be evaluated according to the values of all relevant variables at the time the error occurs, and control will be transferred to the resultant label. Since this delayed evaluation is used, the following simple code will not work:

```
     for i← 1 step 1 until 5 do
          RUN.ERROR (GOOF[i], 'RWR0' + i);
```

At the time of any RWRn error, control will be transferred to GOOF[i], not GOOF[n]. Thus only simple labels should be used for L1; the programmer may use designational expressions to take advantage of this delayed evaluation of L1, but should so with great caution.

3. Run errors in RUNERROR are fatal; that is, a call of the form:

```
RUNERROR(L1, 'RUNR');
```

will not permit recovery from errors in RUNERROR.

4. The switch set by supplying the parameter ERROR.PRINTING is kept in a single cell (-;42); it does not invoke the stacking machinery applied to the error-recovery switches.

<u>ALARMS</u>

The message RUN.ERROR - RUNR will result if

1. An attempt is made to have more than 20 error-recovery switches set at any time, or if

2. The designational expression L1 is undefined at the time of the error; this will occur if the actual parameter is a switch element and the subscript is out of bounds at the time the error occurs.

ALGOL Symbolic Library

PROCEDURE SPECIFICATION

REAL PROCEDURE SIM (N, A, B, Y);

VALUE N, A, B; REAL A, B; INTEGER N; ARRAY Y;

REFERENCE

Algorithm 84, Comm. ACM, April 1962

PURPOSE

SIM determines the approximate numerical value of the definite integral of a continuous function:

$$SIM = \int_A^B f(x)dx$$

RESTRICTIONS

1. The number N of subdivisions must be even.

2. The values Y[I] of the function must be given for equally spaced values of the independent variable on the interval [A,B] of integration.

3. Y[0] = f(A) and Y[N] = f(B)

4. In the program which calls SIM, the actual parameter y corresponding to the array Y must be declared as ARRAY y[0:n], where n is the actual parameter corresponding to N.

METHOD

Simpson's well-known formula for numerical integration is used. If A = X[0], X[1],...,X[N-1], X[N] = B are equally spaced points of subdivision of the interval [A,B] and Y[I] is the value of the function f at the point X[I], I=0,...,N then

$$\int_A^B f(x)dx = \frac{B-A}{3N} (Y[0] + 4Y[1] + 2Y[2] + \ldots + 4Y[N-1] + Y[N])$$

TIME and ACCURACY

The approximate execution time required by SIM is

$$T = 2N*10^{-4}$$

where T = time in seconds

N = number of subdivisions.

The accuracy obtained depends very much upon the function being integrated and the number of subdivisions.

Standard Function

PROCEDURE SPECIFICATION

real *procedure* SIN (X); *value* X; *real* X;

PURPOSE

SIN computes the sine of X, where X is in radians and may be either positive or negative.

METHOD

The method is described on page A-1 of "1604 routine".

ALARMS

RUN ERROR - SIN␣ $|X| > 2,097,151$

For values beyond this point the algorithm breaks down.

TIMING and ACCURACY

The result is produced in 1.08 milliseconds. The relative error is less than $5_{10}-11$.

AL.SIN.2

ALGOL Relocatable Library

PROCEDURE SPECIFICATION

```
procedure SLEW(LFT, RECORD.NUMBER,EOF);
     value LFT, RECORD.NUMBER;
     integer LFT, RECORD.NUMBER; label EOF;
```

PURPOSE

Reading or writing a tape record is delayed by the amount of time required to position the tape to the specified record. SLEW enables the programmer to position a tape while carrying on other computation. The use of SLEW can thus substantially decrease the running time of a program which uses magnetic tape.

METHOD

A call for SLEW will initiate a tape motion to record record.number of logical file lft and will return control to the user's program. Computation or other input/output operations may then continue until the execution of a call for DISC.READ or DISC.WRITE with the parameters lft and record.number. At that time, computation will cease until the tape is positioned and the tape operation completed. Thus it is possible to overlap tape-slewing with useful computation. If logical file lft is on the disc, no action will be taken and the user's program will continue.

USAGE

The statement

```
SLEW(3, 40, eof);
```

will slew to record 40 of the Type 1 RETAP records.

ALARMS

SLEW will exit to the label EOF if RECORD.NUMBER is greater than the maximum record associated with logical file LFT.

Run-error messages are:

SLW1 - LFT is not in the range $2 \leq \text{LFT} \leq 20$

SLW2 - logical file LFT is undefined

TIMING

The time to position a tape to record M, if it is positioned at record N, is about

$$(8 + | M-N - 1 | * 20) \text{ ms.}$$

The actual time required to instruct the tape unit, which is the time spent in SLEW itself, is about 2 ms.

Standard Function

PROCEDURE SPECIFICATION

real **procedure** SQRT (X); **value** X; **real** X;

PURPOSE

SQRT computes the square root of X.

METHOD

For details see page M-1 of "1604 routines".

ALARMS

RUN ERROR SQRT if $X < 0$.

TIMING and ACCURACY

The result is produced in 1.7 milliseconds.

The relative error is less than $1_{10}-12$.

ALGOL Relocatable Library

PROCEDURE SPECIFICATION

procedure AND.FILE (USER, PROG, LF.TYPE, BUFF.LOC, ERR.EXIT);
value PROG, LF.TYPE;
string USER; *integer* PROG, LF.TYPE, BUFF.LOC; *label* ERR.EXIT;

PURPOSE

AND.FILE associates Logical File Type number LF.TYPE with the AND program ("file") specified by USER and PROG. Thus an ALGOL user may define (or redefine) entries in the Monitor Logical File Table during the ALGOL run as well as during execution of the AND system. See Chapter 6g for more information.

PARAMETERS

USER = The AND "User" number for the file, an eight character alphanumeric string; this string may appear either as an 8-character string constant (e.g. 'PH33WW01') or as the name of the first of two successive elements of a logic array which contain the string as value (see example in Chapter 6g).

PROG = The AND "Program" number of the file.

LF.TYPE = The Logical File Type to be associated with this AND file; LF.TYPE must lie in $1 \leqq \text{LF.TYPE} \leqq 19$.

BUFF.LOC = An array element which is the first element of a vector of at least 320 words; this array will be changed by AND.FILE.

ERR.EXIT = A label to which AND.FILE will exit if any of its parameters are improper or if the designated AND file does not exist in the AND Directory.

METHOD

The AND file "USER, PROG" is looked up in the AND Directory and Logical File Type LF.TYPE is associated with it. The 320 locations

starting at BUFF.LOC are used as a buffer to read the AND Directory. After the call of AND.FILE, this space is available for other uses. See Chapter 6g for an example.

ALARMS

If AND.FILE is called to look up an AND file under a man number differing from the man number appearing on the Job Card, then the corresponding Logical File Type will be marked as "read-only".

Any of the following errors will cause AND.FILE to print an appropriate error message and exit to the label ERR.EXIT without assigning the Logical File Type.

1. Usage number is improperly formed or not in AND Directory.
2. Program number is out of range or not in AND Directory.
3. Designated file is empty (contains 0 words) and can therefore be neither read nor written.
4. Logical File Type is < 1 or > 19.
5. Selected AND file is on a tape which is temporarily unavailable.
6. The AND instruction DONT has not been executed, so the AND Directory cannot be read by this CP.
7. An attempt has been made to access another man's file which has been marked "secret" by the AND instruction SECRET. (See the AND writeup.)

PROCEDURE SPECIFICATION

procedure AND.CALL (IMAGES.IN.SCRATCH);
value IMAGES.IN.SCRATCH;
integer IMAGES.IN.SCRATCH;

PURPOSE

AND.CALL may be used to enter the AND system to operate on card images or binary information written into the AND Scratch Area by an ALGOL program.

METHOD

The AND system is loaded and entered by a special entry, which sets the AND Scratch pointer, σ, to (IMAGES.IN.SCRATCH) + 1. If Scratch contains binary information, IMAGES.IN.SCRATCH should be set to $\dot{\downarrow}$(Number of words of binary information + 20)/21. Once loaded, AND will operate as usual, reading and executing AND instruction cards from the Hollerith file which is currently open for input.

Before executing the AND.CALL procedure, the ALGOL program must have loaded the AND Scratch Area (Logical File Type = 1 or 2) with card images using HOLLER.OUT or DISC.WRITE, or binary information using DISC.WRITE.

It is not possible to return to the original ALGOL program after the execution of AND.CALL, since the AND system overlays all of core; however, ALGOL may be called from AND to perform a new translation.

EXAMPLE: An ALGOL program has generated 20 card images in AND Scratch, at which point it executes an AND.CALL (20). The AND system is then loaded and entered with $\sigma \leftarrow 21$, and AND begins reading cards. If the first card contains the AND instructions:

```
AN      FILE 6/0 ; DUMP; DONE;
```

then the 20 images generated by the ALGOL program will be dumped as the 20 cards of AND file 6.

PROCEDURE SPECIFICATION

procedure HOLLER.IN (LF.TYPE);
value LF.TYPE ; *integer* LF.TYPE;

or

procedure HOLLER.IN (LF.TYPE, POINTER);
value LF.TYPE; *integer* LF.TYPE, POINTER;

PURPOSE

HOLLER.IN sets the Monitor's card source pointers for Hollerith card reading to an image in the disc/tape file with Logical File Type = LF.TYPE. After a call of HOLLER.IN has selected a file in this way, executions of "E" or "W" primaries in ALGOL read statements (or an explicit call for |16 in machine language) will bring in successive card images from this file. See Chapter 6g for more information.

PARAMETERS

LF.TYPE = The Logical File Type to be selected as Hollerith input source, replacing the previous source. LF.TYPE in the range: $1 \leqq$ LF.TYPE $\leqq 19$ represents one of the pre-assigned files or an AND file; LF.TYPE = 0 always represents the "primary" Hollerith card source, the one in effect when the ALGOL program began execution.

If $1 \leqq$ LF.TYPE $\leqq 19$, then:

POINTER = An arithmetic expression whose value is the serial number of the first image to be read from file LF.TYPE. If POINTER is omitted, then the first image to be read will be the one after the last image read during the last previous selection of this LF.TYPE by HOLLER.IN; however, if HOLLER.OUT or HOLLER.OVER have also selected the same LF.TYPE, then the rule is more complicated; see Chapter 6g. If file LF.TYPE has not been previously

selected during this run, POINTER = 1 will be assumed if POINTER is omitted.

If LF.TYPE = 0 then:

POINTER = An arithmetic variable which will be set equal to the serial number of the next card to be read from the previously selected file.

METHOD

See Chapter 6g for complete discussion.

ALARMS

RUN ERROR - HIN1: LFT < 0 or LFT > 19.

RUN ERROR - HIN2: An attempt has been made to select the primary source (LFT = 0) when it is already selected.

RUN ERROR - HIN3: The Logical File Type specified has no file assigned to it.

RUN ERROR - HIN4: POINTER < 1, or beyond the physical End-of-File.

ALGOL Relocatable Library

PROCEDURE SPECIFICATION

DISC.READ (NWDS, FIRST.LOC, LF.TYPE, BLK.NO, EOF.EXIT);

DISC.WRITE (NWDS, FIRST.LOC, LF.TYPE, BLK.NO, EOF.EXIT);

value NWDS, LF.TYPE, BLK.NO;

label EOF.EXIT; integer NWDS, LF.TYPE, BLK.NO;

PURPOSE

DISC.READ reads information from disc or tape into an array in core memory; DISC.WRITE writes the contents of an array in core memory out onto disc or tape. These routines allow the user to move binary and Hollerith information in bulk between core memory and disc/tape files. See Chapter 6g for a complete discussion.

PARAMETERS

NWDS = Number of machine words to read or write.

FIRST.LOC = An array element which is the ALGOL name of the first word in core memory to read or write.

LF.TYPE = Logical File Type (explained in Chapter 6g).

BLK.NO = Number of the first 320-word block to be read or written (the first block of the file has block number 0).

EOF.EXIT = Label through which the routine will exit if the physical End-of-File is reached during the operation.

METHOD

DISC.READ reads into core memory exactly NWDS words, where NWDS is not necessarily a multiple of 320. DISC.WRITE will write enough extra words (generally "garbage") to complete the last 320-word physical record. These routines initiate direct input/output transmission between core memory and the tape or disc unit; there is no buffering.

Attempting to read or write beyond the physical End-of-File will cause the routine to complete the operation up to the End-of-File, and

then exit through the EOF.EXIT label.

ALARMS

DISC.READ AND DISC.WRITE give the same error messages, as follows:

RUN ERROR - RWR1: The specified Logical File Type has not been pre-assigned or assigned by AND or by AND.FILE.

RUN ERROR - RWR2: A negative number of words has been called for.

RUN ERROR - RWR3: The FIRST.LOC address lies outside user's memory.

RUN ERROR - RWR4: LF.TYPE is out of range: $1 \leq LFTYPE \leq 19$.

RUN ERROR - RWR5: DISC.WRITE has been asked to write on a file which is marked "read-only".

PROCEDURE SPECIFICATION

procedure HOLLER.OUT (LF.TYPE);
value LF.TYPE ; integer LF.TYPE;

OR

procedure HOLLER.OUT (LF.TYPE, POINTER);
value LF.TYPE; integer LF.TYPE, POINTER;

PURPOSE

HOLLER.OUT sets switches which cause subsequent Hollerith print images to be appended to the disc/tape file with Logical File Type = LF.TYPE. The output file selected by HOLLER.OUT is referred to as the "secondary output" file. When a secondary output file has been selected by HOLLER.OUT, each execution of an "E" or "W" instruction in an ALGOL print statement (or a "W" instruction in an ALGOL read statement) copies the first 80 characters of the output line, followed by a four character serial number, into the secondary output file as the next 84-character image.

A Boolean variable PRINT.OR.NOT, global to HOLLER.OUT, determines whether or not the print image will be transmitted to the "primary" output destination(s) (the LP12 printer and/or teletype output file) in addition to the secondary file.

PARAMETERS

LF.TYPE = The Logical File Type to be selected as the secondary output file. If another secondary file is currently selected, then a Hollerith End-of-File image will automatically be written as the next image in this latter file, and its image pointer will be saved with the Logical File Type. Then:

If IF.TYPE lies in the range 1 ≦ LF.TYPE ≦ 19, the file associated with LF.TYPE will be selected as the secondary output file.

If LF.TYPE = 0, no secondary file will be selected; all subsequent print images will be directed to the primary output destination, regardless of the truth value of PRINT.OR.NOT.

If 1 ≦ LF.TYPE ≦ 19 then:

POINTER = An arithmetic expression whose value is the serial number of the first image to be appended to the secondary file. If POINTER is omitted, then the first image written will be the next after the last image written during the most recent selectic of this LF.TYPE by HOLLER.OUT; however, if HOLLER. or HOLLER.OVER have also selected the same LF.TYPI then the rule is more complex (see Chapter 6g). LF.TYPE has not been previously selected during tl run, POINTER = 1 will be assumed if POINTER is omitted.

If LF.TYPE = 0 then:

POINTER = An arithmetic variable which will be set equal to the serial number of the EOF image written into the previously selected file.

PRINT.OR.NOT = A Boolean variable global to HOLLER.OUT. Th user must declare the variable PRINT.OR.NOT in hi outer-most block. Its value can be changed by th main program to turn primary output printing on (true) or off (false) without calling HOLLER.OUT again. PRINT.OR.NOT is ignored when no secondary output file is selected.

METHOD

See Chapter 6g for complete discussion.

ALARMS

RUN ERROR - HOT1: LF.TYPE < 0 or LF.TYPE > 19.

RUN ERROR - HOT2: LF.TYPE = 0, and there is no secondary file currently selected.

RUN ERROR - HOT3: Logical File Type LF.TYPE has no file assigned to it.

RUN ERROR - HOT4: POINTER < 1, or beyond the physical End-of-File.

RUN ERROR - HOT5: Attempt to write on file marked read-only (i.e., another user's file).

USE

Before an ALGOL program using HOLLER.OUT terminates execution, it must call HOLLER.OUT(0) to write out the last physical record of the last-used secondary output file; a Hollerith End-of-File image will be written onto the file at this time. If the programmer fails to do this, he will lose up to the last 16 card images.

PROCEDURE SPECIFICATION

procedure HOLLER.OVER (LF.TYPE);
value LF.TYPE;
integer LF.TYPE;

OR

procedure HOLLER.OVER (LF.TYPE, POINTER);
value LF.TYPE;
integer LF.TYPE, POINTER;

PURPOSE

HOLLER.OVER is basically the same as HOLLER.OUT. While HOLLER.OUT appends Hollerith output images to a file, however, HOLLER.OVER allows the user to alter ("*over*write") any individual card images in the middle of an existing AND file (which may have been created originally by AND or by HOLLER.OUT).

METHOD

HOLLER.OVER differs from HOLLER.OUT in two ways:

(1) HOLLER.OVER does not write a Hollerith End-of-File image onto the previously selected secondary record. HOLLER.OVER writes an End-of-File image *only* if an attempt is made to write on the last image of the physical file.

(2) HOLLER.OVER reads each block of the file as it currently exists into the output buffer before new images are entered.

PARAMETERS

See AL.5.HOLLER.OUT.

ALARMS

See AL.5.HOLLER.OUT.

CHAPTER 6a

ALGOL-20 Card Format and Keypunching Conventions

ALGOL programs should be punched in the following form:

```
                   111 1 1 111 11 2 222 2222223333333333...R
Column →  12 3 45678 9012 3 4 567 89 0 123 4567890123456789   M

Language
   ↓
 WHAT    WH   .LOC.      F   OP.    M    Addr,Index;Comment..
 ALGOL   AL   ..................ALGOL text..................
 system  SY   ..................system text.................
 comment CO   ....................comment...................
```

The teletype TAB table for these language fields is as follows:

```
Tab →   1      2      3      4      5

Language field
   ↓
 AL     4      7      10     13     16
 WH     4      15     20     24     40
 SY     4   tabs after the first are taken as characters and scanned.
 CO     no tabs - tabs are taken as characters.
 Tabs taken as characters are printed as '=' on the LP-12.
```

A blank language field (columns 1 and 2) is interpreted as AL or WH, as determined by the most recent appearance of AL or WH in the language field. A blank language field on the first card is interpreted as AL.

System cards give special instructions to the ALGOL translator. The system instructions are described in Chapter 4.

Comment cards are printed as part of the translation listing but are otherwise ignored. They may be freely inserted for purposes of documentation.

Normally, the programmer may use columns 4-72 for his ALGOL program; however, there is a system statement with which he may change the right-hand margin from 72 to any column between 40 and 80, as described in Chapter 4. The columns beyond the right margin may be used for comments, etc.

The translator will ignore all columns to the right of a double vertical bar (||). This allows the programmer to use the rest of the card for comments. (Double vertical bars in strings will not invoke this convention.)

In ALGOL-20, certain constructions may not be split between the end of one ALGOL text card and the beginning of the next, since the translator always supplies an imaginary blank column between the right-hand margin of one card and column 4 of the next:

(1) An *identifier* may not be continued from one card to the next. The implied blank will terminate an identifier which ends at the right margin of the card.

(2) The *combination characters*

:= ¬> ¬< ¬= 8L 8R 8F

must be punched with the two characters in adjacent columns of the same card.

In addition, *alphanumeric strings* may not be split between two cards. One reason for this restriction is to provide better error recovery if a quote is accidentally omitted. A long alphanumeric string instruction in a READ, PRINT, or PUNCH statement may simply be closed by " ', " (quote comma) after the last character of the string punched on the card and reopened with another quote on the next card.

The ALGOL program may be punched in any format the programmer desires, subject only to the above restrictions. There may be one statement per card or a single statement extended over many cards or many statements on a single card. Thus, a consistent indentation rule which aligns each *end* with its *begin* can be used to advantage to show clearly the structure of the program. (See also Chapter 6c.)

CHAPTER 6b

ALGOL-20 Error Messages

Translation Errors

The ALGOL-20 translator prints numerical error codes to indicate syntax errors in the source program. The translator prints the error code just below the last card image it has scanned, with an arrow '↑' pointing to the last character scanned on this image. Since each card is scanned once from left to right, the '↑' pointer will generally be ahead (i.e., to the right) of the actual error.

Errors are broadly categorized as Phase I errors (0 through 69), Phase II errors (70 through 95), miscellaneous errors (96 through 99), Subscan errors (100 through 109), and System errors (110 through 125). System errors are those occurring on SY cards. Subscan and Phase I errors are purely syntactic and are discovered in the process of scanning the source program cards. Phase II errors are discovered at a later stage in the translation process when the actual machine code of the object program is generated. The miscellaneous errors are those which indicate a possible problem within the translator. Any listing containing such an error should be brought to the attention of the User Consultant or a member of the Computation Center staff immediately.

The translator attempts to "recover" from each error if possible, so that many independent errors may be found in one pass through the computer. However, any Phase I error as well as errors marked with * in the error list will cause all succeeding Phase II errors to be ignored. To call the programmer's attention to the fact that subsequent Phase II errors are being ignored, the translator will print "NOTE 6" (see the section on notes below) as Phase II is turned off.

Any error prevents execution of the compiled program.

Notes

The Algol-20 translator prints notes on the program listing to call the programmer's attention to syntactic constructions which are acceptable but questionable, or constructions which are possibly caused by an error. Notes do not prevent execution of the compiled program.

Translation Errors

Phase I Errors (Each of these errors terminates Phase II.)

0: The program does not start with begin.
1: A statement starts with an illegal character or an illegal reserved word.
2: A statement starts with an identifier followed by an illegal character.
3: In an expression an operand was expected and was not found.
4: In an expression a binary operator was expected and was not found. (Possibly caused by a semicolon missing after the preceding statement.)
5: A "]" does not have a matching "[".
6: An array element has been used illegally.
7: A ":" has appeared incorrectly.
8: A "←" or ":=" has appeared incorrectly.
9: A ")" does not have a matching "(".
10: A "," has appeared incorrectly.
11: then has appeared without if.
12: else has appeared without then.
13: Characters are still in the stack after a ";" or an end.
14: A procedure statement is followed by other than end, else, or ";".
15: for is not followed by an identifier.
16: The for variable is not followed by a "←" or ":=".
17: step has appeared without for.
18: until has appeared without step.
19: while has appeared without for.
20: do has appeared without for.
21: go to is not followed by an identifier or "(" or if.
22: go to if...then...is not followed by else.
23:
24: An obscure error in a go to statement.
25: An impossible error after begin. ("⊢" is not the second element in the stack. See Error 98.)
26: own is followed by something other than <type>.
27: An array declaration does not specify subscript bounds.
28: The identifier list of a declaration is not followed by a ";".
29: switch is not followed by an identifier.
30: The identifier of a switch declaration is not followed by a "←" or ":=".
31:
32: procedure is not followed by an identifier.
33: A procedure identifier is not followed by "(" or ";".
34: A formal parameter list is not followed by ")".
35: The ")" following a parameter list is not followed by a ";".
36: The identifier list in a specification is not followed by a ";".
37: An identifier did not follow the "," in an identifier list.
38: The illegal construction "then if" has occurred.
39: A switch with more than one subscript position has been used.
40: The value part of a procedure declaration was not followed by a ";".

41: The name of a permanent subroutine (such as "SIN") is not followed by "(".
42: There is an extra "," or else a missing ":" in an array declaration.
43: More begin's than end's have occurred when the end-of-file is reached.
44: Impossible - see Error 98.
45: max or min is not followed by "(".
46: In an array declaration the identifier list is not followed by "[".
47: Array specifier has subscript bounds, which it should not.
48: library is not followed by <type> or procedure.
49:

Phase I Errors (format and name statements) (Each of these errors terminates Phase II.)

50: A reserved input/output word is not followed by "(" .
51: A format list element starts with an illegal character. (Should be "<" or "→" or "$" or identifier).
52: "→" is missing: i.e., a replicator was expected but not found.
53: for is missing after "$".
54: "→" is not followed by "$" or an identifier.
55: ")" or ">" is not followed by ")" or ",".
56: A name statement or format statement is not followed by end, else or ";".
57: A replicator is not followed by "(" or "<".
58: "<" or "," is followed by an illegal character.
59: An integer is followed by an illegal character.
60: A format instruction is not followed by ">" or ",".
61: An illegal prefix to a numeric primary has been used.
62: An illegal numeric primary has been used.
63: "." appears in a numeric primary in a read statement.
64: In a numeric primary, E, F or S is not followed by an integer.
65:
66:
67:
68:
69:

Phase II Errors (Only those errors marked "*" turn off Phase II.)

*70: A reserved word which is not yet available has been used.
71: A label has been used but not defined. (The name of the label is printed prior to this error message)
72: An identifier has been used but not declared.
73: An identifier has been declared twice in this block.
74: An identifier in the value list is not a parameter.
75: An identifier which has been used as a procedure has not been declared to be one.
76: A subscripted identifier has not been declared to be an array or switch.
77: The program is too long.
78: A procedure identifier which is not a function designator has been used in an expression.

* Turns off Phase II.

79: An identifier which has been used as a switch has not been declared to be one.
80: An array identifier has been used without subscripts.
81: Too many index variables have been declared.
82: A label or array or switch has been called by value.
83: An identifier in a specification list is not a parameter.
84: In a procedure declaration a parameter is not specified.
85: In a procedure declaration a parameter is specified twice.
86: A procedure, switch or label appears on the left of a ":=" or "←".
*87: The W2 stack is too full.
88: More than 100 relocatable <u>library</u> <u>procedures</u> have been declared.
89: A constant has been used in place of an identifier, e.g., 33[k].
*90: A subscripted for variable has been used (this is not yet available in ALGOL-20).
*91: The next-command pointer is less than the base of the program.
92:
93:
94:
95:

Miscellaneous Errors

96:
97: A possible translator error - bring listing to Janet Fierst at the Computation Center.
98: Impossible: bring your listing to A. Evans at the Computation Center.
99: Same as 98.

Subscan Errors

100: A card column contains an illegal combination of punches.
101: Too many abcons or adcons have been used (numerical constants and alphanumeric string constants).
102: Too many decimal points appear in a number.
103: Too many "$_{10}$"s appear in a number.
104: An error has appeared in a parameter delimiter comment: ")<any string not containing:>:(".
105: An illegal bar ("|") variable has been used.
106: A constant has been used which is too large to fit into a <u>real</u> variable.
107: A "$_{10}$" is followed by something other than "+", "-", or <digit>.
108: A string goes over the end of a card.
*109: The symbol table has been exceeded.
110:
111:
112:
113:
114:

* Turns off Phase II.

System Statement Errors

115: An abcon system statement has occurred after code has been compiled.
116:
117: An abcon system statement has requested more space than there is in user memory.
118:
119: An illegal SY card has occurred. (This may be caused by a LIBRARY card after the symbolic library has been released.)
120: The library procedure nesting exceeds 5.
121:
122: WHAT has been called after it has been released.
123: An illegal segment statement has been used.
124: An SY LIBRARY card has asked for a routine not in the symbolic library.
125: A library procedure declaration has named a routine not in the relocatable library.

Notes

Note 1: end comment convention was used on preceding card. That is, everything was ignored up to ";", end, or else.
Note 2: A function designator has been used as a procedure statement.
Note 3: In an arithmetic or boolean expression, the construction if...then if has occurred. This is syntactically illegal but unambiguous, and is therefore accepted by the translator.
Note 4: An arithmetic (boolean) (designational) expression has been used where a simple arithmetic (boolean) (designational) expression should have been used.
Note 5: In a designational expression, the construction if...then if has occurred. This is syntactically illegal but unambiguous.
Note 6: Phase II has been turned off.
Note 7: The construction if...then for...do...else... which is legal in ALGOL 60 but illegal in ALGOL 62 has been used.
Note 8: TAB appears as a character.
Note 9: Fifty errors have been found on a single card; compilation has been terminated.

RUN ERRORS

During its execution, an ALGOL-20 program will call upon various run-time subroutines. Since there are usually restrictions placed upon the use of these routines, there is a mechanism provided to warn the user when the restrictions have been violated. It is the violation of these restrictions which is referred to as "run error".

The user may receive his warning in either of two ways: If he takes no action otherwise, an error message will be printed out as part of his output listing which will identify the error and the part of the program in which the error took place. The run will then be terminated. On the other hand, however, the programmer may provide for a "recovery procedure" by calling on a subroutine named RUN.ERROR. If he does this, an occurrence of the error will cause a transfer to a statement in his program which he has designated. The programmer may then provide error recovery as he deems appropriate. If the programmer has provided for error recovery from a given error type, he has the option of either getting or not getting the normal diagnostic printout. See Chapter 5.RUNERROR for a description of the recovery procedure.

The normal error printout consists of the two lines

RUN ERROR - eeee
COMMAND IN ERROR - ccccc

followed by fourteen lines of diagnostic output. Here <u>ccccc</u> is the (octal) location of the command which caused the error (or the command which called the subroutine in which the error was detected) and <u>eeee</u> is a mnemonic error code. By comparing <u>ccccc</u> with the octal addresses printed on the left side of the compilation listing, the programmer may determine the particular statement or declaration whose execution caused the error. Error codes for library procedures are given in the ALARMS section of the procedure description in Chapter 5. All other error codes are listed on the next page. The fourteen lines of diagnostic information refer to the seven commands before the faulty one, the faulty command, and the six commands following. On each line, there are four fields: the location, the command in octal, the command in semi-mnemonic form, and the contents of the word whose address is in the command. This information is useful to anyone familiar with G-20 machine coding in analyzing the error. It is of interest to the average user only in that it may be shown to the User Consultant.

Error Code	Meaning
ADRP	address--opcode fault
CFLG	command flag error
EXP	EXP (X) called with X > 160.116998
EXPO	exponent overflow
LN	LN (X) called with X ≤ 0
RAD1	upper < lower in a bound pair in an array declaration
RAD2	declared arrays exceed available space
READ	An error has occurred in reading a data card. (see below)
SIN	The argument to SIN or COS exceeds 8↑21
SQRT	SQRT (X) called with X<0.
TIMR	monitor detected errors
X↑A1	X = 0 and A≤0 in X↑A
X↑A2	A * LN (X) > 160.116998 in X↑A
X↑A3	X ≤ 0 and A not integer valued in X↑A

When an error is detected by the read subroutine, the "RUN ERROR-READ" message will be preceded by a printout of the data card containing the error, an arrow (↑) pointing to the erroneous column, the column number, and one of the following auxiliary messages:

1) $$ - CARD READ	An End-of-File mark has been reached: i.e., the program has attempted to read more data cards than are in the "deck".
2) NO CARD READ	A read statement has attempted to scan characters from a card image before an "E" or "W" primary has been executed to read a card.
3) IMPROPER NUMBER	An ill-formed number has been scanned; e.g., too many decimal points, + or - signs, or $_{10}$'s have been found.
4) ILLEGAL SYMBOL	A character which cannot be part of a number (i.e., not a digit, ".", "+", "-", or "$_{10}$") has been scanned by a numeric instruction. This error message is suppressed by the suffix "N".

AL.6b.8

CHAPTER 6c

Printing of the Compiled Program

Consistent indentation of each level of nesting of subordinate and compound ALGOL constructions is useful in writing a clean, readable program. The maximum possible indentation of the program as punched onto cards is limited by the width of the card. However, the printed image is 21 columns wider than the card, so the system statement INDENT has been provided to let the programmer take advantage of the extra printer width to get more indentation of the printed source program. Normally, the compiled ALGOL code is printed in the format

PRINT columns	123	4567	8....12	13,14	15...83	84...104	105...120
Contents		blank	address	blank	text	blank	comments
CARD columns	123				4....72		73.....88

Using the system statements INDENT and RIGHT MARGIN (see Chapter 4), the programmer may change the number of card columns scanned as text and also may change the print columns in which this text appears. In general, the compiled code is printed as follows, where K stands for the indent constant and RM stands for the right margin:

PRINT columns	123	4567	8....12	13...14+K	15+K...11+K+RM	12+K+RM...32+RM	33+RM...120
Contents		blank	address	blank	text	blank	comments
CARD columns	123				4...........RM		RM+1.....88

The address printed is that of the first instruction generated by the line of ALGOL text.

AL.6c.2

CHAPTER 6d

Privileged Identifiers

In addition to the reserved identifiers, ALGOL-20 includes a set of "privileged" identifiers which have built-in meanings. These identifiers can be used with their built-in meanings without being declared; they are, in effect, declared by the translator in a block head outside the outer-most block of the program. Therefore, if the programmer does not wish to use one of these identifiers in its privileged meaning, he may simply ignore the fact that it is privileged and declare and use it as he would any non-reserved identifier. If a privileged identifier is redeclared within an inner block, it resumes its privileged meaning as soon as the end of that inner block is passed.

The currently available privileged identifiers in ALGOL-20 are described below. As additional privileged identifiers become available, they will be described on sheets which can be added to this chapter.

ACC

ACC is a symbol denoting the accumulator, which may appear on the left side of an assignment statement. Thus

ACC:= A[i, j, k+3];

will fetch an array element to the accumulator. ACC is of particular use in setting the accumulator before executing a piece of WHAT code. This mechanism should always be used when accessing array elements or formal parameters called by name for use in WHAT code. No error will be detected if ACC appears other than on the left side of an assignment statement, but such uses will usually cause spurious results.

CLOCK

CLOCK is an integer-valued function designator which is called with a single parameter. The value of "CLOCK(0)" is the elapsed time in seconds since the Name-Time card was read. "CLOCK(V)" is the elapsed time in seconds minus the integer value of the parameter V, also in seconds; i.e.,

CLOCK(V) = CLOCK(0) - V

Example:

```
A:  STARTTIME ← CLOCK(0) ;
        .
        .
        .
B:  ELAPSE ← CLOCK(STARTTIME) ;
```

This will store in ELAPSE the elapsed time in seconds between passing label A and passing label B.

DAY

MONTH

YEAR

DAY, MONTH, and YEAR are built-in variables of type logic which are set by the translator to the four-character alphanumeric string representations of the current day, month, and year, respectively. The format is best defined by example: On the 9th of April, 1962, we have

```
DAY = '␣09␣'
MONTH = 'APR␣'
YEAR = '62␣␣'
```

The statements:

```
NAME ( DAY, MONTH, YEAR ) ; PRINT (<12A, E>) ;
```

would print a line containing:

```
09 APR 62
```

DEBUGPRINT

DEBUGPRINT is a procedure with an arbitrary number of parameters which prints the values of its parameters in a simple rigid format. Its parameters may be any arithmetic or Boolean expressions. Arithmetic values are printed in the format <+.11ZL> ; i.e., in scientific floating point notation with 11 significant figures. Boolean values are printed as:

TRUE for true

$+.00000000000_{10}+00$ for false

Values are printed four per line; the first value printed by each call of DEBUGPRINT starts on a new line.

EPSILON

EPSILON is a built-in real constant, whose value is the smallest positive number which can be represented in the G-21.

EPSILON = 8^{-63}

$\approx 1.274473528903_{10}-57$

HALT

HALT is a parameterless procedure. Execution of a HALT statement terminates the run-time execution of the program and returns control to the monitor. Thus, executing a HALT statement is equivalent to letting control pass the final end of the program.

EXAMPLE: if X > $_{10}8$ then HALT ;

INFINITY

INFINITY is a built-in real constant, whose value is the largest positive number which can be represented in the G-21.

INFINITY = $(8^{14} - 1) * 8^{63}$

$\approx 3.450873173389_{10}69$

PAGES

PAGES is an integer procedure with no parameters. Its value is the total number of pages which have been completely printed since the job card was printed. If the printer is positioned to the page which contains the job card information, the value of PAGES is zero.

PAUSE

Executing this parameterless procedure invokes the monitor PAUSE mechanism, |27, in the usual way. When the program is subsequently restarted, PAUSE will return to its calling point. For further details, see the description of |27 in the Monitor Description: THEM THINGS.

PR1NT

PR1NT is a procedure with one parameter which controls the output of printed information on a teletype. (Note the spelling: the third character is a one.) The compilation listing and execution-time output are always printed on the on-line printer, if a positive number of pages is requested on the Job Card. PR1NT sets a monitor switch which determines whether execution-time output will also be typed on the teletype which originated the program. If the program did not originate at a teletype, then PR1NT will simply be ignored.

PR1NT (1) : Sets switch so output will be on the teletype and printer.

PR1NT (0) : Sets switch so output will not be on the teletype, only on the printer.

The status of the switch can be changed as often as desired; when an ALGOL program begins execution, it is set to PR1NT (0) unless zero pages have been requested on the Job Card, in which case, it is set to PR1NT (1).

TIME

TIME is a real procedure with no parameters, whose value is the time of day in seconds, starting at midnight.

CHAPTER 6e

Machine-Dependent Features

1. Octal Constants

An octal (base 8) constant may be used in any context in ALGOL-20 where a decimal number is allowed; i.e., as a primary in any arithmetic or logic expression. Octal constants have the following syntax:

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

<octalian> :: = <octal digit> | <octalian> <octal digit>

<signed octalian> :: = <octalian> | + <octalian> | - <octalian>

<left-justified octal constant> :: = 8L <octalian>

<right-justified octal constant> :: = 8R <octalian>

<floctalian> :: = <octalian> | <octalian>.<octalian> | <octalian>. | .<octalian>

<power of 8> :: = $_{10}$<signed octalian>

<floating octal constant> :: = 8F <floctalian> | 8F <power of 8> | 8F <floctalian> <power of 8>

<logical octal constant> :: = <left-justified octal constant> | <right-justified octal constant>

<octal constant> :: = <floating octal constant> | <logic octal constant>

Despite this syntax, the translator does not treat the digits 8 and 9 in octal constants as erroneous but will interpret them as $10)_8$ and $11)_8$, respectively. Thus 8R495 = 8R515.

Local octal constants (8L and 8R) are considered to be of type logic and so are always accessed in logic mode. Floating octal constants (8F) are considered to be of arithmetic type, and are always accessed in arithmetic mode.

The character-pairs 8L, 8R and 8F are treated by the translator as single entities and <u>must</u> be punched in adjacent columns of the same card, without intervening blanks.

The value of a floating octal constant is determined by concatenating the floctalian as an octal number and multiplying it by the appropriate power of 8, treating the number which follows the $_{10}$ as an octal integer. For example:

$$8F_{10}10 = 8\uparrow 8$$

$$8F11_{10}-5 = 9*8\uparrow -5$$

The value of a left (right) justified octal constant is determined by prefixing (suffixing) to the octalian enough zeros to give eleven octal digits. This number is then concatenated and stored as a 32-bit logic word. Since eleven octal digits require thirty-three bits for representation, the leftmost bit of the leftmost octal digit is lost. Thus, 8L4 = 0 and 8L7 = 8L3, a "3-flag".

2. String Constants

Alphanumeric strings of not more than four characters may be used as constants in an ALGOL-20 program. Such a string, converted to a set of G-20 internal characters and stored right-justified in a 32-bit logic word in the abcon region, is treated as a logic octal constant. Since the G-20 internal character for a blank ('␣') is zero, the following string constants all have the same value:

'A' = '␣A' = '␣␣A' = '␣␣␣A'

String constants may be used in any context in which octal constants are allowed.

Examples:

1. <u>if</u> nextchar = ':' <u>then</u> stringval ← ';'
2. 'ABCD' ∧ 8L377

(The value of the expression in example 2 is 'A␣␣␣')

3. <u>if</u> (x∧ 8R377) = 'A' <u>then</u>

(This tests whether the right-most character of x is 'A'. Note that the parentheses are needed, since without them, the meaning is <u>if</u> x∧ (8R377 = 'A') <u>then</u> which is always false.)

3. Bit-Manipulation Operations

In addition to arithmetic, Boolean, and designational expressions, ALGOL-20 snytax includes "logic expressions" which perform bit-by-bit logic operations on 32-bit G-20 logic words. A logic expression may include any of the following operands:

1. Logic constant: octal constant or string constant
2. Variable, simple or subscripted, of type logic
3. Function designator of type logic
4. Boolean primary (and, therefore, any Boolean expression in parentheses)
5. Arithmetic primary (and, therefore, any arithmetic expression in parentheses)

A Boolean primary used as a logic operand is interpreted as one of the two 32-bit logic words:

8R 37777777777 = 32 one bits for true, or
8R 0 = 32 zero bits for false.

Each kind of logical operand (except number 5 above, arithmetic primary) will always be fetched from memory with a "logic access", rather than a "numeric access"; for example, a CAL command will be used to fetch a logic variable into the accumulator. When a logic variable or function designator forms the left-part of an assignment statement, then an STL command will perform the assignment. Therefore, an assignment statement of the form

<logic variable> ← <arithmetic expression>

will truncate the absolute value of the expression modulo 2^{32}. An STL command is also used for any temporary store of a logical subexpression (except an arithmetic primary) within a complete logical expression.

Any of the following three logical operators may appear in a logic expression:

¬ (complement logic: unary)
∧ (extract logic: binary)
∨ (unite logic: binary)

Each of these operators performs the same operation simultaneously and independently in each of the 32-bit positions of its operand(s). If a bit = 1 represents the Boolean value true while a bit = 0 represents false, then the logic operators ¬, ∧, and ∨ can be considered to perform the Boolean operations ¬, ∧, and ∨, respectively, in each bit position.

The operators +, -, *, and / may also appear in a logic expression. Each of these operates in the usual way, considering its logical operands (except for arithmetic primaries) as 32-bit integers.

The complete syntax for logic constructs is given below:

```
<logic constant> :: = <string constant> | <logic octal constant>
<logic primary> :: = <logic constant> | <logic variable> | <logic function> |
                          <Boolean primary> | (<logic expression>) |
                          <arithmetic primary>
<logic factor> :: = <logic primary> | ¬<logic primary>
<logic term> :: = <logic factor> | <logic term> ∧ <logic factor>
<simple logic expression> :: = <logic term> | <simple logic expression> ∨
                                             <logic term>
<logic expression> :: = <simple logic expression> | <if clause>
                        <simple logic expression> else <logic expression>
```

4. Half Variables

A variable of type half behaves exactly as one of type real except that it contains fewer digits (about 6 instead of about 12). Since half variables take only one location in the machine and real variables take two, it is possible to save storage space by declaring large arrays to be of type half instead of real, if the loss of significance can be tolerated.

5. Index Variables

Simple variables of type index are stored in G-20 index registers. Eventually, efficient code will be compiled for index variables, using the G-20 index register commands. For the present, however, index is of interest only in connection with machine language assembly code ("WHAT") within an ALGOL-20 program. Index variables behave exactly as do integer variables.

CHAPTER 6f

Segments

ALGOL programs which are too long to fit into core memory may be divided into subportions, called segments. Each segment will be stored on tape after it is compiled and may then be called into core memory as it is needed.

Each segment area on tape holds 10240_{10} words. If a segment contains more than this, it will be stored on successive segment areas. (The number of such areas needed is the second parameter to the SY Segment statement.)

A segment is an ALGOL-20 program which includes a SEGMENT system statement. Each segment must be a complete ALGOL-20 program; that is, it must have a set of matching *begin*'s and *end*'s and declarations of all identifiers which are used in the segment.

After a segment has been compiled (i.e., after the *end* which matches the first *begin* is found), the code which has been compiled and any relocated subroutines will be written onto tape as the specified segment. ALGOL will then continue reading cards, expecting to find another program; this program may also be a segment. Compilation terminates after a program which is not a segment is processed; control is then transferred to the first statement of this program.

If in a program (or segment) the statement LINK(i) appears, segment i will be loaded and control will be transferred to its outermost *begin*.

It is convenient to be able to communicate data in core memory between different segments. However, linking from one segment to another involves exiting from the outermost block of the current segment, thus making all variables declared in the segment undefined. To overcome this, the same block of storage locations is used in all segments for *own* arrays and *own* scalars. Thus, if identical declarations of *own* arrays and *own* scalars are made in different segments, all these values may be transmitted from one segment to another. *All* other scalars and arrays are undefined when a new segment is loaded; these may be communicated by means of the procedures DISC.WRITE and DISC.READ.

AL.6f.2

ERRATA

Two minor features of the HOLLER.OUT and HOLLER.OVER routines do not yet function as described in Chapter 6g. These features are as follows.

(1) HOLLER.OUT (and -- .OVER) do not yet use the complete AND serial numbering system for serial numbers > 9999; instead, these routines now supply purely numeric 4-digit serial numbers, modulo 10^4. See Section I of Chapter 6g.

(2) Page headings, run error messages, and DEBUG.PRINT output <u>will</u> appear in an open secondary output file as well as in the primary output file. See Section V B of Chapter 6g.

Both these features will soon be corrected to correspond to Chapter 6g.

CHAPTER 6g

ALGOL DISC/TAPE ROUTINES

CONTENTS

I. Introduction - Files and Records.

II. Logical File Types.

- A. Preassigned Logical File Types.
- B. Addressing Disc/Tape Files via Logical File and Relative Addresser Tables.

III. The Binary File Routines - DISC.READ and DISC.WRITE.

- A. Parameters and Call.
- B. Examples of Use.
- C. An Algorithm for Buffered Reading.

IV. The AND.FILE Routine.

V. The Hollerith File Routines.

- A. Introduction.
- B. Primary vs. Secondary Files.
- C. Opening and Closing Hollerith Files.
- D. Card Image Pointer.
- E. Parameters and Call of HOLLER- Routines.
- F. The Operation of HOLLER.IN.
- G. The Operation of HOLLER.OUT and HOLLER.OVER.

I. INTRODUCTION - FILES AND RECORDS.

ALGOL-20 includes a set of relocatable library subroutines for storing binary and Hollerith data in the G-21's disc memory or on magnetic tapes. This chapter explains these routines and their relationship to the AND system and to certain parts of the Monitor. A knowledge of AND is assumed, although no knowledge of the Monitor is required to understand this chapter.

Disc and tape space is divided into segments of consecutive storage called files. A file in turn is generally subdivided into a number of sections with similar (or identical) format, called logical records. The user may subdivide a file into logical records in any way he pleases; for example, the logical records could be of varying lengths, or they could all have a fixed length (e.g., 237 words each). Files are recorded on disc and tape in units of 320 words, called physical records or blocks. If the length of the logical records is not a multiple of 320 words (the physical record size), then transmission of logical records to and from disc and tape files requires the use of buffer areas in core memory to pack and unpack logical records into physical records.

The G-21 Monitor and the AND System use a standard format for files of 80-character "card images" of Hollerith information. A file with this standard format is called a "Hollerith File", and always consists of 21-word logical records. Each logical record contains 84 characters, packed four-characters-per-word; the last four characters form a Hollerith serial number, the kth image in the file having serial number k (except see the AND System description, page 4.407 of the User's Manual, for the numbering system beyond card 9999).

The G-21 Monitor includes a "card read routine", |16, which can unpack 21-word logical records of a Hollerith input file from a 320-word internal buffer area in core memory. The relation of |16 to ALGOL programs is explained more fully in Section VB.

Each file is assigned space on disc or tape in units of complete blocks, i.e., in multiples of 320 words. The space allotted to a file has a fixed length and therefore an end, called its physical (or binary) End-of-File. ("End-of-File" will be abbreviated to "EOF" in this chapter.) Disc/Tape routines will detect an attempt to read or write beyond the physical end of a file, and will notify the programmer of the End-of-File condition, for example, by returning through a special exit. Thus, a program which processes a file may use the EOF condition as a signal to transfer out of the main processing loop.

The routines for reading and writing disc and tape may be divided into two sets: the binary file routines, and the Hollerith file routines.

1) Binary File Routines: DISC.READ, and DISC.WRITE.

These routines are very efficient for moving large quantities of data between ALGOL arrays in core memory and bulk storage (i.e., disc or tape files). They assume no logical record structure for the file, but read and write in units of complete 320-word physical records. Files read with DISC.READ and written with DISC.WRITE will be referred to as "binary files", since no logical record structure is assumed by these routines; however, such binary files could contain any mixture of binary numbers and Hollerith strings arranged by the programmer (see the examples in Section III). The programmer using the binary file routines may wish to provide buffering in his ALGOL program to pack and unpack logical records which are not a multiple of 320 words in length; see Section III for a complete discussion.

2) Hollerith File Routines: HOLLER.IN, HOLLER.OUT, and HOLLER.OVER.

These routines allow ALGOL format READ, PRINT, and NAME statements to be used to read and create Hollerith files. For example, the HOLLER.IN ("HOLLERith INput") routine sets switches in the Monitor to cause "E" and "W" instructions in READ statements to take successive images from a selected Hollerith File on disc or tape. Then the full machinery of NAME and READ statements may be used to scan the images character-by-character

for Hollerith strings and numbers, to convert them to binary, and to store them into ALGOL variables. The HOLLER.OUT ("HOLLERith OUTput") and HOLLER.OVER ("HOLLERith OVERwrite") routines provide the corresponding ability for creating Hollerith files, image-by-image, using NAME and PRINT statements. HOLLER.OUT appends images to the end of a file, while HOLLER.OVER is used to alter images of an existing Hollerith file. Since all Hollerith files have identical format, a file created by HOLLER.OUT (or -.OVER) may subsequently be edited by the AND System. See Section V for a complete explanation of these routines.

Since a Hollerith file is composed of 21-word logical records, $\frac{320}{21} \cong 15.24$ card images fit into each physical record; a file with binary length of B blocks can contain at most $\downarrow(\frac{320 \text{x} B}{21}) - 1$ card images. However, a Hollerith file may contain fewer than this maximum number of images. The end of Hollerith information is indicated by a Hollerith End-of-File image which normally appears immediately after the last image currently in the file.

A Hollerith End-of-File image is distinguished by the presence in columns 1 and 2 of two "lower-case dollar signs", G-20 character code $_8165$, This code is the G-20 internal representation of the (+, -, 8, 9) punch combination which always appears in columns 1 and 2 of a job card; thus, each job card serves as an EOF card for the preceding job in the card reader. The rest of the EOF image contains a message indicating which routine wrote it there; for example, the EOF image: "$$ ALGOL END OF FILE" is written by HOLLER.OUT. An EOF image from the card reader is blank except the two lower-case dollar signs.

It is usually important to have an EOF image in a Hollerith file since, for example, the Monitor card read routine |16 checks for the Hollerith EOF image but not for the physical EOF. Hence, if the programmer is using the EOF condition to terminate his program when it reaches the end of reading a Hollerith file, he must have an EOF image in the file. Hollerith files created by AND always contain EOF images; those

created by HOLLER.OUT will have an EOF image if the programmer properly "closes" the file - see Section V.

The disc/tape routines DISC.WRITE, HOLLER.OUT, and HOLLER.OVER allow an ALGOL user to write directly onto any of his AND files -- i.e., any files which are listed under his man number in the AND Directory. However, these routines never change the dump count of the file being written. Therefore, a user who writes files using these ALGOL subroutines and saves the file for later runs must be aware of the possibility (against which the dump count protects AND users) that all AND files may be set back one or more days as a result of hardware failure. To protect himself against this possibility, the programmer may want to keep his own dump count somewhere in the file, increment the count when he writes on the file, and check for the value he expects before reading or writing. If the AND file is written only once and thereafter is only read, or is used only as a temporary ("scratch") file during a run, then there is no need for a dump count.

The user should also note that the AND files are stored not only on the disc but also on two permanently-mounted magnetic tapes. It requires approximately <u>three</u> <u>minutes</u> of computer time to traverse one of these tapes end-to-end. An ALGOL user who attempts to access more than one file on the same tape can easily waste huge amounts of computer time moving ("slewing") the tape back and forth between files. On the other hand, any number of files on disc can be accessed "randomly", i.e., without wasting time slewing. Note: a file is on disc if its First Block Number in the Directory is less than 21000. A new AND instruction will shortly be available to force a file to be created or dumped onto the disc rather than onto one of the AND tapes; the programmer will therefore have a convenient way of avoiding severe tape slews due to the ALGOL disc/tape routines.

II. LOGICAL FILE TYPES.

A. PREASSIGNED LOGICAL FILE TYPES.

The ALGOL disc/tape routines refer to particular files by an integer value between 1 and 19 called the Logical File Type ("LF Type"). Some of the Logical File Types have been pre-assigned to certain fixed files in the CIT system: e.g., the AND Scratch Area. However, any of the 19 Logical File Types may be assigned to any permanent AND file for the duration of the run, either by the AND System or by the ALGOL library procedure AND.FILE. Normally, a user should use AND to set up all LF Types for all files he will need, before the ALGOL translator is called to compile and execute his program. In some cases, the user will need to assign AND files to LF Types during his ALGOL run; the library procedure AND.FILE is provided for this purpose (see Section IV).

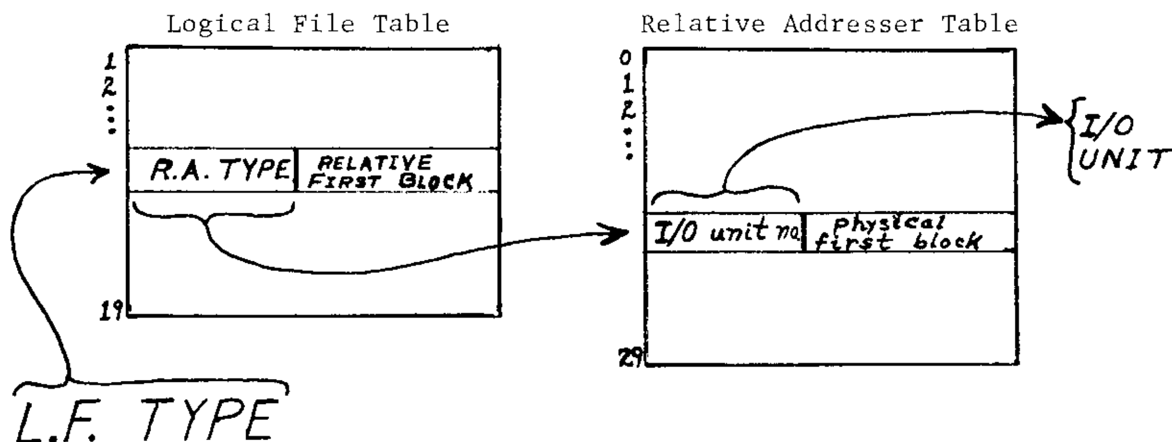
The LF Types currently pre-assigned are as follows:

LF Type	Meaning	Length (blocks)	Length (card images)
1	Current effective beginning of AND Scratch Area.	(see below)	(see below)
2	Physical beginning of AND Scratch Area	3072	46811
3	Retap 1 Records (same as AND Scratch Area).	3072	46811
4	Retap 2 Records (the second half of AND Scratch).	1536	23405
5	Retap 3 Records (on a permanently mounted system tape)	≧10000	≧ 152381
6	Comp Center Records (NOT available to user, unless specifically allocated by CC staff!).	-----	-----

In some circumstances (specifically, the execution of a "RUN, AND, ..." instruction) the AND System moves the effective beginning of the AND Scratch Area past the end of information already in Scratch. AND defines Logical File Type 1 to be the effective beginning of the Scratch Area when AND loads and executes the ALGOL system. Logical File Type 2, however, is always defined as the fixed physical beginning of the Scratch Area. It is recommended that LF Type 1 rather than LF Type 2 be used to refer to AND Scratch so that "recursive" AND runs can be performed without changing the operation of the ALGOL program.

B. ADDRESSING DISC/TAPE FILES VIA THE LOGICAL FILE AND RELATIVE ADDRESSER TABLES.

The correspondence between a particular LF Type and actual block addresses on the physical input/output devices is established by two tables in the Monitor: the Logical File table, and the Relative Addresser table. The ALGOL programmer is thereby removed by two levels of generalized (or "symbolic") addressing from the physical input/output devices. The process of finding a block on the physical device can be visualized by the following picture:



The LF Type is used to select an entry in the Logical File Table which specifies: (1) a Relative Addresser Type ("RA Type") and (2) the first block of the file, relative to the first block of the region defined by the RA Type. This RA Type is used in turn to select an entry in the Relative Addresser Table, which specifies: (1) the (logical) number of an actual I/O unit and (2) the first physical block number of that RA Type region on that unit. This addressing process can be summarized by the following ALGOL statements: to find block number B of the file with Logical File Type LF.TYPE, the Monitor uses:

```
U ← IO.UNIT [RA.TYPE [LF.TYPE]] ;
P ← PHYSICAL.FIRST.BLOCK [RA.TYPE [LF.TYPE]]
        + RELATIVE.FIRST.BLOCK [LF.TYPE] + B
```

where U is the logical I/O Unit number, and P is the physical block number on that unit. The logical unit number U is converted to a physical I/O unit number through another Monitor Table to achieve flexibility in the use of the physical tape drives; this last logical-physical correspondence is irrelevant to the programmer, however.

The Relative Addresser Table (discussed on page 14 of the Monitor "|routine" write-up "THEM THINGS") provides a global allocation of space on tape and disc, generally the same for every program run. The Relative Addresser Table is changed in two circumstances: (1) The Computer Operations staff may change the global allocation of space on tapes and disc, or may convert the entire operation to tapes when the disc is "down"; and (2) a user may specify a "User Tape" on his job card, replacing one (or more) of the standard system Relative Addresser entries by entries pointing to his own magnetic tape, only for the duration of his run. If a User Tape replaces a Relative Addresser Type to which a Logical File Type is assigned, then the meaning of the Logical File Type and the length of the file change correspondingly, but only for the duration of the run.

III. THE BINARY FILE ROUTINES: DISC.READ AND DISC.WRITE.

A. PARAMETERS AND CALL.

DISC.READ will read information from disc or tape into an array in core memory; DISC.WRITE will write the contents of an array in core memory out onto disc or tape. These routines allow the user to move binary and Hollerith information in bulk between core memory and disc/tape files. Their parameters are as follows:

DISC.READ(NWDS, FIRST.LOC, LF.TYPE, BLK.NO, EOF.EXIT);

DISC.WRITE(NWDS, FIRST.LOC, LF.TYPE, BLK.NO, EOF.EXIT);

Here:

NWDS = Number of machine words to read or write.

FIRST.LOC = An array element which is the ALGOL name of the first word in core memory to read or write.

LF.TYPE = Logical File Type (explained in Section II).

BLK.NO = Number of the first 320-word block to be read or written (the first block of the file has block number 0).

EOF.EXIT = Label through which the routine will exit if the physical End-of-File is reached during the operation.

Starting at the block number BLK.NO of the tape/disc file with Logical File Type LF.TYPE, the routines will read or write as many blocks as are needed for the number of words NWDS. DISC.READ reads into core memory exactly NWDS words, where NWDS is not necessarily a multiple of 320. DISC.WRITE will write enough extra words (generally "garbage") to complete the last 320-word physical record. These routines initiate direct input/output transmission between core memory and the tape or disc unit; there is no buffering.

Attempting to read or write beyond the physical End-of-File will cause the routine to complete the operation up to the End-of-File, and

then exit through the EOF.EXIT label.

If an AND file assigned to a Logical File Type (by AND.FILE, or by the AND system) belongs to a programmer different from the man making the run, then this LF Type will be marked "read only". Calling DISC.WRITE (or HOLLER.OUT, or HOLLER.OVER) to write on this file will cause a run error; thus, a user can write only on his own files. Both DISC.READ and DISC.WRITE print the same run-error messages, as follows:

RUN ERROR - RWR1: The specified Logical File Type has not been pre-assigned or assigned by AND or by AND.FILE.

RUN ERROR - RWR2: A negative number of words has been called for.

RUN ERROR - RWR3: The FIRST.LOC address lies outside user's memory.

RUN ERROR - RWR4: LF.TYPE is out of range: 1 ≦ LFTYPE ≦ 19.

RUN ERROR - RWR5: DISC.WRITE has been asked to write on a file which is marked "read-only".

B. EXAMPLES OF USE.

Assume the following declarations:

<u>real</u> <u>array</u> X[0:20, 1:100] ;

<u>integer</u> <u>array</u> I[0:1000] ; <u>logic</u> <u>array</u> L[1:300] ;

Then the subroutine call:

DISC.READ (4200, X[0,1], 1, 23, EOF);

reads 4200 words into the X array in core memory from the file with Logical File Type 1 (the AND Scratch Area), starting from block 23 of this file. Each element of the real array X occupies two machine locations and X[0,1] is the first element of X; therefore, this call will exactly fill the X array. There is no check in DISC.READ (or DISC.WRITE) for overflowing an array; if this call were changed to read more than 4200 words, the extra words would be stored into locations beyond the end of the X array, presumably clobbering another array.

Similarly, the entire I array may be written on records 40 through 43 of AND Scratch by the procedure call

DISC.WRITE (1001, I[0], 1, 40, EOF);

Since each integer variable is stored in one G-21 memory location, 1001 words are needed for the 1001 elements of the array. A little more than three blocks are needed for this WRITE operation; the fourth block will be filled up with whatever happens to be in memory in the locations following I[1000].

These routines may be used to read or write alphanumeric strings; such strings should be stored in core in arrays of type <u>logic</u>. Each word of the string contains four characters. Thus, the call:

DISC.WRITE (300, L[1], 1, 0, EOF);

will write a string of 4 x 300 = 1200 characters from the array L.

C. AN ALGORITHM FOR BUFFERED READING.

In all the above examples, each <u>logical</u> record read or written was assumed to start at the beginning of a <u>physical</u> record. If the logical records contain significantly fewer than 320 words, however, this simple assignment of logical to physical records is wasteful of both time and space on tape or disc. If disc/tape space is important and if the logical records are short, the user should include buffer routines in his ALGOL program to pack and unpack logical records from each physical record. The buffer area would be an ALGOL array whose length was a suitable multiple of 320.

For example, the procedure declaration below describes the algorithm for buffered reading from a binary file. The algorithm reads BUFF.SIZE blocks at a time into a buffer array:

BUFF [1:320*BUFF.SIZE].

Note that BUFF is a "dynamic <u>own</u> array" (see page AL.2.14 of ALGOL-20 Manual); if this algorithm is used in an ALGOL-20 program, a numerical

value must be substituted for BUFF.SIZE before the program is compiled, or else BUFF must be declared globally to the procedure. Although the formal parameter A in BUFFERED.READ is a logic array, an array of any type may be substituted as an actual parameter; however, if a real array is substituted, then the actual parameter substituted for N must be doubled.

```
procedure  BUFFERED.READ (N, A, LF.TYPE, EOF.EXIT);
     value  N, LF.TYPE;
     integer  N, LF.TYPE;  logic array A;  label  EOF.EXIT;

comment  Each call of BUFFERED.READ moves the next logical record of N
logic words into variables A[1], ..., A[N] of a logic array A.  If this
process empties the internal buffer array BUFF, DISC.READ is called to
refill the buffer from the file with Logical File Type LF.TYPE.  If the
physical EOF is encountered before the logical record is obtained,
BUFFERED.READ exits to the label EOF.EXIT.
     Calling BUFFERED.READ with LF.TYPE < 0 initializes the record
number and buffer pointer to the beginning of block 0 of the Logical File
Type -LF.TYPE, and moves the first n-word logical record of the file;

begin
     own logic array  BUFF [1:320*BUFF.SIZE];
     own integer  PT, BLK.NO; integer  I, ULIM;
     if  LF.TYPE < 0  then
        begin  BLK.NO ← 0 ; PT ← 320*BUFF.SIZE  end  initialization;
     I ← 1;
MORE:
     ULIM ← min (N, 320*BUFF.SIZE - PT);
     for  I ← I  step  1  until  ULIM  do
        A[I] ← BUFF[PT + I];
           comment  move words until N words moved or buffer empty;
     if  ULIM < N  then
        begin comment  BUFF is empty and more words are needed;
           DISC.READ (320*BUFF.SIZE, BUFF[1], abs (LF.TYPE), BLK.NO,EOF.EXIT);
              BLK.NO ← BLK.NO + BUFF.SIZE;
              PT ← PT - 320*BUFF.SIZE;
              go to  MORE
        end of refill;
     PT ← PT + max (N, 0);
end  BUFFERED.READ ( );
```

IV. THE AND.FILE ROUTINE

The relocatable library procedure AND.FILE allows an ALGOL program to define a Logical File Type as a particular AND file. AND.FILE is called with the following parameters:

AND.FILE(USER, PROG, LF.TYPE, BUFF.LOC, ERR.EXIT);

where:

USER = The AND "User" number for the file, an eight character alphanumeric string; this string may appear either as an 8 character string constant (e.g., 'PH33WW01') or as the name of the first of two successive elements of a logic array which contains the string as value (see example below).

PROG = The AND "Program" number of the file (not including the dump-count).

LF.TYPE = The Logical File Type to be associated with this AND file; LF.TYPE must lie in the range: $1 \leqq LF.TYPE \leqq 19$.

BUFF.LOC = An array element which is the first element of a vector of at least 320 words.

ERR.EXIT = A label to which AND.FILE will exit if any of the parameters are improper or if the designated AND file does not exist in the AND Directory.

The effect of calling AND.FILE is to look up the AND file under (USER, PROG) in the AND Directory, and assign it to Logical File Type LF.TYPE. The 320 locations starting at BUFF.LOC are used as a buffer to read the AND Directory; after the call of AND.FILE, this space is available for other uses.

It is important to observe that AND.FILE reads but never writes on the AND Directory; thus, the AND Directory will never be changed as a result of execution of AND.FILE. This implies that:

(1) the file being looked up must already exist; it will not be created by AND.FILE if it does not exist;

(2) operation of AND.FILE will never change either the physical (i.e., binary) length of the AND file, or the most-recent-access date in the Directory, or the Dump Count of the file.

In general, a run using the AND system will be required to create an AND file or to increase its length. There is an AND instruction "CREATE" for this purpose.

The user is warned that if he accesses an AND file only in his ALGOL program via the AND.FILE procedure, never in AND itself, then the latest-access-date for the file will never be updated. As a result, the file will eventually be "frozen", i.e., moved to a history tape and deleted from the current AND records. On the other hand, setting up a Logical File Type during an AND run does update the access date of the file in the AND Directory; hence it is generally preferable to set up L.F. Types during the AND run rather than to use AND.FILE.

An example of a call of AND.FILE would be:

```
integer array  BUFF [1:320] ;
label GLUG;
logic array  USER[1:2] ; integer I, PROG.NO;
:
name (I → 2(USER[I]) , PROG.NO.) ;
read (<E, 8A, 8D>) ; comment read user number and program number
                      from a data card;
:
AND.FILE(USER[1] , PROG.NO, 12, BUFF[1] ,GLUG) ;
:
GLUG:  PRINT (<'NO SUCH FILE EXISTS', 2E>) ; HALT ;
```

This program will read an 8 character user number string (e.g., 'S236JP01') and an integer program number from a data card, and look

up the corresponding file in the AND Directory. If the file exists, it will become Logical File Type 12; if not, the statement labeled "GLUG" will be executed. The 320 words BUFF[1]...BUFF[320] will be used to read the AND Directory.

If AND.FILE is called to look up an AND file under a man number differing from the man number appearing on the Job Card, then the corresponding Logical File Type entry will be marked "read-only". Calling DISC.WRITE, HOLLER.OUT, or HOLLER.OVER to write on a read-only AND file will cause a run error; thus, a user can use the ALGOL disc/tape routines to write on his own AND files only.

AND.FILE checks carefully the validity of its parameters. Any of the following errors will cause it to print an appropriate message and exit to the label ERR.EXIT without defining the Logical File Type. A subsequent attempt to operate on this Logical File Type with one of the other disc/tape routines will cause a run error. The error conditions detected by AND.FILE are:

1. Usage number is improperly formed or not in AND directory.
2. Program number is out of range or not in the AND directory.
3. Designated file is empty (contains 0 words) and can therefore be neither read nor written.
4. Logical File Type does not satisfy $1 \leqq$ L.F. Type ≤ 19.
5. Selected AND file is on a tape which is temporarily unavailable.
6. AND instruction DONT has not been executed, so the AND Directory cannot now be read by this CP. See AND System Write-up.
7. Attempt has been made to access another man's file which is marked "secret". See AND System Write-up.

V. THE HOLLERITH FILE ROUTINES

A. INTRODUCTION

The DISC.READ and DISC.WRITE routines move binary and Hollerith information in bulk between core memory and disc/tape files. The "HOLLER-" routines, HOLLER.IN, HOLLER.OUT, and HOLLER.OVER, on the other hand, provide buffered input and output of Hollerith files using ALGOL format READ, PRINT, and NAME statements.

Calling HOLLER.IN selects a Hollerith file as the source of card images for READ statements; each subsequent execution of an "E" or "W" instruction will read the next image from the selected file into the format read buffer, where it can be scanned character-by-character. Similarly, HOLLER.OUT (or HOLLER.OVER) will cause each "E" or "W" format instruction in a PRINT statement to output columns 1 through 80 of the 120-column print buffer plus a sequence number in columns 81 to 84, as the next image of a selected Hollerith output file. HOLLER.OUT (and - .OVER) supply consecutive integer serial numbers, equal to the ordinal numbers of the images in the file. HOLLER.OUT appends images to a file, while HOLLER.OVER may be used to alter images in the middle of an existing Hollerith file.

B. PRIMARY vs. SECONDARY FILES

When an ALGOL program begins execution under the G-21 Monitor, there is always one Hollerith input file and one Hollerith output file; these are referred to as the "primary" input and output files, respectively. The primary input file may be a teletype input file, an AND file, the AND Scratch Area, or a physical deck in the card reader, for example. The primary output file may be the LP12 printer, a teletype output file, or both. The HOLLER- routines designate auxiliary input and output files, referred to as "secondary files".

When an "E" or "W" format instruction is executed in a READ statement, the ALGOL format routine calls the Monitor card read

routine ("|16") to supply a Hollerith card image. |16 has implicit parameters which indicate whether the image is to come from the next physical card in the card reader, or from the next 21-word logical record in a particular Hollerith file on disc or tape. In the latter case, |16 transmits to the (84-character) ALGOL READ buffer* the card image which is the next 21-word logical record in a 320-word input buffer area in core memory. If this image was the last complete image in the 320-word buffer, then |16 automatically reads the next 320-word block from disc or tape into its buffer area, in anticipation of supplying the next image. (Note: this description has omitted some details which are unnecessary to an understanding of the HOLLER- routines; further information on Monitor routines may be found in Section 2.3 of the CIT User's Manual.)

The function of HOLLER.IN is to set the source parameters of |16 to take Hollerith images from a particular file. Calling |16 either implicitly with "E" or "W" format READ instructions, or explicitly in machine language with WHAT code will read successive images from the file selected by HOLLER.IN. It should be remembered that <u>any</u> routine in the user's program whose effect is to read "cards" will ultimately call |16, and will therefore get images from a secondary file if HOLLER.IN has been called.

Similarly, when the ALGOL format routine executes an "E" or "W" instruction in a PRINT statement, or a "W" instruction in a READ statement, it calls the Monitor Hollerith output routine ("|11") to transmit the print line to the primary and/or a secondary output file. The function of HOLLER.OUT (and -.OVER) is to set the secondary destination parameter in |11 to a particular secondary output file. The

---

*Note: the "<u>ALGOL READ buffer</u>", part of the format READ mechanism of ALGOL, contains exactly one 84-character card image to be scanned by format instructions; it should not be confused with the 320-word <u>input buffer area</u> used by |16 to unpack 21-word logical records from 320-word disc/tape blocks.

images sent to a secondary file are collected in a 320-word buffer in core memory; when this buffer is filled, its contents are automatically written onto the next physical block on tape/disc. Even if the programmer calls |11 directly in WHAT code, |11 will transmit the print line to the primary and/or secondary output files as determined by HOLLER.OUT (or -.OVER). On the other hand, ALGOL run-error messages, page headings, and output from DEBUG.PRINT are transmitted only to the primary output file, never to a secondary output file.

C. OPENING AND CLOSING HOLLERITH FILES

The process of selecting a Hollerith file with HOLLER.IN, HOLLER.OUT, or HOLLER.OVER will be referred to as "opening" the file; the "deselection" of a file will be referred to as "closing" that file. Closing a secondary output file writes the last 320-word block from the core buffer onto the file on disc or tape. The user of HOLLER.OUT or HOLLER.OVER is cautioned that the system does not automatically close a secondary output file when the ALGOL program terminates. The user must call HOLLER.OUT(0) or HOLLER.OVER(0), as appropriate, to close the file and write out the last block, or he will lose up to 16 card images.

Because input can be read from only one source at a time, there can be only one open input file at once: either the primary input file, or a secondary file selected by HOLLER.IN. On the other hand, the same output line may be transmitted simultaneously to more than one output file, so one secondary output file (selected by HOLLER.OUT or HOLLER.OVER) may be open simultaneously with the primary output file, if the user so chooses.

D. CARD IMAGE POINTER

The Hollerith card read routine |16 has an input card image pointer which contains the serial number of the next card image to be read from the current file (unless the input file is a physical card deck, in which case the card image pointer is undefined); each call

of |16 increments this pointer automatically. The Hollerith print routine |11 has a corresponding output card image pointer which contains the serial number of the next card image to be output to a secondary output file, but is undefined if no secondary output file is open. Note that the output card image pointer is associated only with a secondary file, and has no necessary connection with the serial number of the primary output file. If a secondary output file is open, then each call of |11 converts the value of the image pointer to 4 Hollerith characters and stores it as the serial number in columns 81-84 of the image (note: this serial number is also stored into columns 81-84 of the print line sent to the primary file); then the image pointer is incremented by 1.

HOLLER.IN sets the input card image pointer when it selects a new input file; first, however, the previously selected file is closed, and the previous value of the card image pointer is saved in a pointer temporary associated with the previous LF Type. Similarly, when HOLLER.OUT or HOLLER.OVER selects a new secondary output file, it closes the previous file and saves the previous value (if any) of the output card image pointer in the pointer temporary associated with the LF Type of the previously selected secondary file, before setting the output card image pointer to a new value. The programmer can reselect ("reopen") any secondary file which was selected earlier in the run, either for reading or writing, in such a way as to restore the card image pointer to the value saved in the pointer temporary for that LF Type. It is important to note, however, that there is only one pointer temporary for each LF Type, used for saving both input and output card image pointers. This makes it convenient to read through a file until a certain image is found and then start rewriting the file from that image on.

In addition to the internal pointer temporary mechanism just described, there is provision in the call of the HOLLER- routines for

storing and setting the input and output card image pointers using ALGOL variables designated by the programmer.

When an ALGOL program begins execution, the pointer temporaries for all LF Types (except perhaps 0 and 1) are initialized to 1. Calling AND.FILE to assign a particular LF Type to an AND file initializes the pointer temporary of that LF Type to 1.

It is possible to have the same file, with the same LF Type, open for both Hollerith input and Hollerith output, simultaneously. Furthermore, the same file can be assigned (by the AND system or by AND.FILE) to more than one Logical File Type. This gives the user great flexibility in processing several different parts of the same file at once. However, the user should avoid doing such an operation on a file which is on a tape rather than on the disc, or machine time may be wasted on long tape slews.

It is convenient to think of the primary input file as Logical File Type = 0. For example, when the primary input file is closed by HOLLER.IN and a secondary input file becomes the card image source, the input image pointer for the primary file is saved in the internal pointer temporary associated with LF Type = 0. Calling HOLLER.IN to open LF Type = 0 will reselect the primary input file, after closing the previously selected secondary input file. LF Type = 0 has no relation to the primary <u>output</u> file, which cannot be opened or closed by the HOLLER- routines.

E. PARAMETERS AND CALL OF HOLLER- ROUTINES

The HOLLER- routines may have one or two parameters, as follows:

```
HOLLER.IN(LFT) ;       or      HOLLER.IN(LFT, POINTER) ;
HOLLER.OUT(LFT) ;      or      HOLLER.OUT(LFT, POINTER) ;
HOLLER.OVER(LFT) ;     or      HOLLER.OVER(LFT, POINTER) ;
```

In each case, the formal parameters could be specified by:

<u>value</u> LFT ; <u>integer</u> LFT, POINTER ;

LFT is generally the Logical File Type of the file to be opened for input (HOLLER.IN) or output (HOLLER.OUT or HOLLER.OVER); see the detailed description of each routine below. Any arithmetic expression may be substituted for the LFT parameter.

The meaning of the parameter POINTER depends upon the value of LFT:

$1 \leq LFT \leq 19$: Pointer = the serial number to which the card image pointer will be initialized in the secondary file with LF Type = LFT. In this case, any arithmetic expression may be substituted for the parameter POINTER. If the POINTER parameter is omitted, then the card image pointer will be initialized instead to the value of the internal pointer temporary associated with LFT.

LFT = 0: POINTER is a variable which will be set equal to the previous input or output card image pointer, i.e., the pointer in the file which is being closed; this same pointer value is also stored in the internal pointer temporary associated with the LF Type being closed. In this case, only a variable, either simple or subscripted, may be substituted for POINTER. If POINTER is omitted, then the card image pointer will be stored only in the internal pointer temporary.

The programmer may wish to find out the serial number of the next image to be output to, or read from, an open secondary file; notice that closing and immediately reopening the same Logical File Type will have no effect other than saving the current image pointer in POINTER, if the parameter is present in the statement which closes the file. The user may also wish to skip about in one file, which means that the internal image pointer from a preceding file closure will be lost on a following closure; if the user saves his own copy of the pointer in POINTER when he closes the file, he may reopen the file later at this serial number by using POINTER as parameter.

F. THE OPERATION OF HOLLER.IN

A call of the following form:

HOLLER.IN(LFT); or HOLLER.IN(LFT, POINTER);

opens for Hollerith input the file with LF Type = LFT, after closing for input the file which was previously open. The following steps are performed:

(1) The previous value of the input card image pointer is assigned to the internal pointer temporary for the LF Type of the file previously open for input.

(2) If LFT = 0, (i.e., the primary input file is being selected) then the previous card image pointer is also assigned to the variable POINTER (if the POINTER parameter is present).

(3) The card image source is set to the file with LF Type = LFT:

$1 \leq LFT \leq 19$: opens a secondary input file, and

LFT = 0 : reopens the primary file.

(4) The input card image pointer (i.e., the serial number of the first image to be read from the newly opened file) is set equal to the value of the pointer temporary for file LFT, unless LFT is > 0 (i.e., a secondary file is being selected) and the parameter POINTER is present; in the latter case, the input card image pointer is set to the value of POINTER.

Thus, the ALGOL programmer can always "get hold of" of the input card image pointer of a secondary file by executing: HOLLER.IN(0, HIS.POINTER) to close the secondary input file and store the pointer in the ALGOL variable HIS.POINTER. The user cannot "get hold of" the card image pointer for the primary file, since this file may be a physical card deck for which the pointer is undefined. On the other hand, if the user has arranged it so that the primary file is a particular AND file or one of the files which have preassigned LF Types, then the user

can assign the same file a LF Type and select it as a secondary file with HOLLER.IN; he can then effectively "backspace" and reread such a primary input file.

Whenever the Monitor card read routine |16 transmits a Hollerith EOF image to the ALGOL READ routine or directly to the user, a switch is set in the Monitor; if |16 is called to provide another card image after this EOF switch is set, the Monitor will terminate the program. However, calling HOLLER.IN with L.F. Type = 0 not only closes the secondary file, but also clears this |16 EOF switch. Therefore, the user can read and detect the Hollerith EOF image in a secondary file (by testing for the first two characters being 8R165), close the file, and continue reading images from the primary file or open another secondary file.

HOLLER.IN may produce any of the following run-error messages:

RUN ERROR - HIN1	LFT < 0 or LFT > 19
RUN ERROR - HIN2	Attempt to close a secondary input file when there was no secondary file open for input.
RUN ERROR - HIN3	Attempt to read from a file which was not predefined or defined by AND or AND.FILE.
RUN ERROR - HIN4	Attempt to set the image pointer to a negative card number or beyond the physical End-of-File.

G. THE OPERATION OF HOLLER.OUT AND HOLLER.OVER

A call of the form:

HOLLER.OUT(LFT); or HOLLER.OUT(LFT, POINTER);

will close the secondary file (if any) which was previously open for Hollerith output; if LFT > 0, it will then open the secondary file number LFT for Hollerith input.

If $1 \leq$ LFT ≤ 19, then the following steps are performed:

(1) If there was previously a secondary output file open for Hollerith output, then it is closed in the following manner:

(a) The output card image pointer is assigned to the internal pointer temporary of the LF Type previously open.

(b) A Hollerith EOF image is placed after the last image appended to the file (but the card image pointer is not incremented, so if the file is later reopened at this point then the EOF image will automatically be overwritten by the next image).

(c) The last 320-word block is written from the secondary output buffer in core onto the disc/ tape file.

(2) The secondary output destination is set to the file with LF Type = LFT.

(3) The output card image pointer is set equal to the serial number in the pointer temporary associated with LFT, unless the parameter POINTER is present, in which case, the pointer is set equal to the value of POINTER. The physical block containing this image is read into the secondary output buffer.

If LFT = 0, then the following steps are performed instead:

(1) The secondary output file which was previously open is closed, as described above for the case: $1 \leqq LFT \leqq 19$; if none was previously open, a run error - HOT2 will occur.

(2) If the parameter POINTER is present, then the same pointer value which has been stored in the internal pointer temporary for the previously open file will also be assigned to POINTER.

If HOLLER.OUT is used, the user must declare the Boolean variable PRINT.OR.NOT in his outermost block. If PRINT.OR.NOT = true when |11

is called by an "E" or "W" in a PRINT statement, a "W" in a READ statement, or directly in WHAT code, then the print line will be transmitted to the primary output device as well as to the secondary output file. If no secondary output file is open, PRINT.OR.NOT has no effect. Note that the Boolean value of PRINT.OR.NOT may be set at any time without calling HOLLER.OUT again. When a secondary output file is closed by HOLLER.OUT, a Hollerith EOF image is appended to the file, as described above. The serial number of this EOF image is the pointer to the "next" image which is saved in the pointer temporary and perhaps passed to the user via the POINTER parameter. If the file is subsequently reopened for output with this pointer restored, the End-of-File image will be written over.

Each time a file is opened by HOLLER.OUT, the disc (or tape) block containing the card image at which the file is opened is read into core. However, on succeeding blocks, a read does not precede the write. Thus, HOLLER.OUT may be used to append images to the end of a Hollerith file but not to alter images in the middle of an existing file; the routine HOLLER.OVER should be used for the latter purpose.

WARNING: If an ALGOL program terminates before the secondary output file is closed by a call: HOLLER.OUT(0), as many as the last 16 card images outputted to that file may be lost

If an attempt is made to "HOLLER.OUT" (or "-.OVER") the last card image of the entire physical file, an End-of-File image will be written instead but no error indication will be given; an attempt to write another image, beyond the physical end of the file, will result in an error "HOT3". There should always be an EOF image as the last card image of the physical file; however, closing a file with HOLLER.OUT(0) may write another EOF image earlier.

A page heading or page number which is output to the primary file under control of the |212 and |213 switches (see Chapter 3d) will not appear in the secondary output file. Similarly, run error messages produced by the ALGOL error diagnostic routine as well as output from DEBUG.PRINT will appear only in the primary output file. Thus, DEBUG.PRINT statements can be inserted into a program without changing the number of images sent to the secondary file.

The operation of HOLLER.OVER is similar to HOLLER.OUT with two differences:

(1) HOLLER.OVER does not output an End-of-File image when it closes a secondary file;

(2) HOLLER.OVER reads each block into core before it is altered and written out. Thus, HOLLER.OVER may be used to alter images in the middle of an already existing Hollerith file.

If an error is detected in the HOLLER.OUT or HOLLER.OVER routine, the secondary file will be closed if it was open, and no new secondary file will be opened; however, the last block (up to 16 card images) may be lost.

Both HOLLER.OUT and HOLLER.OVER produce the following run error messages:

RUN ERROR - HOT1	LFT < 0 or LFT > 19
RUN ERROR - HOT2	Attempt to close a secondary file when there was no secondary output file open.
RUN ERROR - HOT3	Attempt to open a LF Type which was not predefined or defined by AND.FILE or AND.
RUN ERROR - HOT4	Attempt to write outside the bounds of the physical file.
RUN ERROR - HOT5	Attempt to write on file marked read-only (i.e., another user's file).

CHAPTER 6h

Storage Allocation

Algol programs go through 3 phases: compilation of the program, loading and relocation of all relocatable subroutines, and running of the program. The same area of memory may be used for different purposes at the three different times. It is necessary to have some understanding of this storage allocation to understand how much space is available for program and for data, and to understand how this space can be expanded. On page AL.6h.2 is a memory map showing storage allocation at the three times. A vertical arrow (↑ or ↓) indicates an area of storage which may expand to the next horizontal line. Opposing vertical arrows ($\overset{\downarrow}{\underset{\uparrow}{}}$) indicates areas of storage which may expand until they meet. A vertical arrow terminated by a horizontal line indicates that the exact upper (or lower) bound of the area is different for each program.

At the end of each Algol program the "words" printed is the total number of words which would be dumped if the program were dumped as a segment - the space from A to B on the memory map. Call this number NWRDS. Since the total space available for this information and for data is

$$C - A = /56720 = 24016_{10} \text{ locations}$$

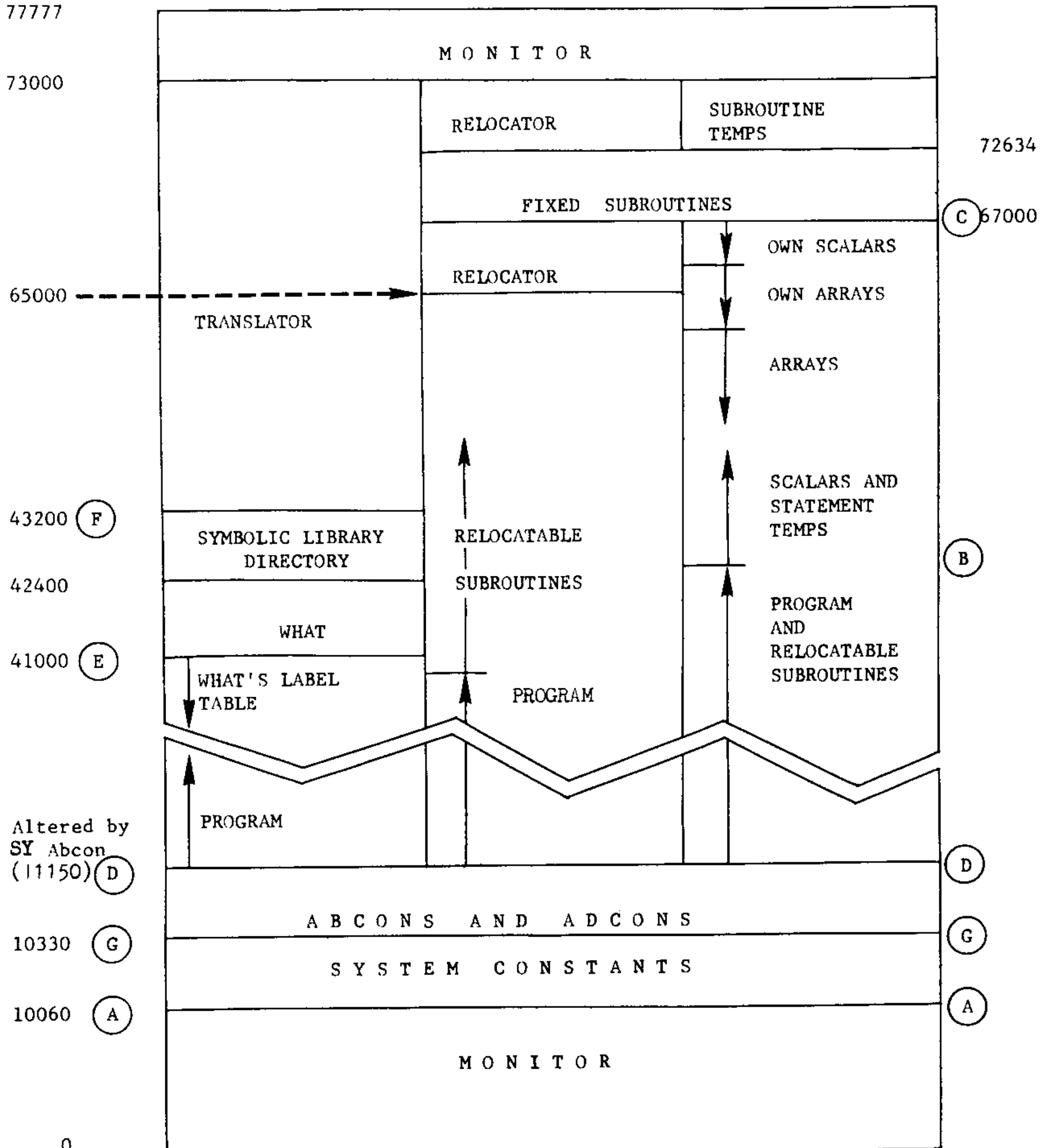
the total space available for data (scalars, arrays, own scalars, own arrays) is 24016_{10} - NWRDS. Let the space required for WHAT labels be NWHAT. Then the space originally available for a program and for WHAT labels is

$$E - D = /27630 = 12384_{10} \text{ locations}$$

The total space permitted the program is thus 12384_{10} - NWHAT. Doing "Release WHAT" and "Release Symbolic library" increases the program space to

$$F - D = /32030 = 13336_{10} \text{ locations.}$$

Changing the number of abcons and adcons changes the location of D and so changes the amount of space available for the program. The system statement "n ABCONS", which is described in detail in Chapter 4, causes D to be set to G + 2n. Since n is initially 200_{10}, D is G + 400_{10}.



(All addresses are octal)

Revised Report on the Algorithmic Language ALGOL 60

PETER NAUR (*Editor*)

J. W. BACKUS	C. KATZ	H. RUTISHAUSER	J. H. WEGSTEIN
F. L. BAUER	J. McCARTHY	K. SAMELSON	A. VAN WIJNGAARDEN
J. GREEN	A. J. PERLIS	B. VAUQUOIS	M. WOODGER

Dedicated to the Memory of WILLIAM TURANSKI

SUMMARY

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

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

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

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

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

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

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

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

CONTENTS

INTRODUCTION
1. STRUCTURE OF THE LANGUAGE
   1.1. Formalism for syntactic description
2. BASIC SYMBOLS, IDENTIFIERS, NUMBERS, AND STRINGS. BASIC CONCEPTS.
   2.1. Letters
   2.2. Digits. Logical values.
   2.3. Delimiters
   2.4. Identifiers
   2.5. Numbers
   2.6. Strings
   2.7. Quantities, kinds and scopes
   2.8. Values and types
3. EXPRESSIONS
   3.1. Variables
   3.2. Function designators
   3.3. Arithmetic expressions
   3.4. Boolean expressions
   3.5. Designational expressions
4. STATEMENTS
   4.1. Compound statements and blocks
   4.2. Assignment statements
   4.3. Go to statements
   4.4. Dummy statements
   4.5. Conditional statements
   4.6. For statements
   4.7. Procedure statements
5. DECLARATIONS
   5.1. Type declarations
   5.2. Array declarations
   5.3. Switch declarations
   5.4. Procedure declarations
EXAMPLES OF PROCEDURE DECLARATIONS
ALPHABETIC INDEX OF DEFINITIONS OF CONCEPTS AND SYNTACTIC UNITS

This report was published simultaneously in the Communications of the ACM, 6, No. 1 (1963), 1-17, the Numerische Mathematik, and the Computer Journal.

INTRODUCTION

Background

After the publication of a preliminary report on the algorithmic language ALGOL,[1,2] as prepared at a conference in Zürich in 1958, much interest in the ALGOL language developed.

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

January 1960 Conference

The thirteen representatives,[3] from Denmark, England, France, Germany, Holland, Switzerland, and the United States, conferred in Paris from January 11 to 16, 1960.

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

April 1962 Conference [Edited by M. Woodger]

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

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

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

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

Questions concerned with the following areas are left for further consideration by Working Group 2.1 of IFIP, in the expectation that current work on advanced pro

* 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.

[1] Preliminary report—International Algebraic Language. *Comm. ACM 1*, 12 (1958), 8.

[2] 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. *Num. Math. 1* (1959), 41–60.

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

gramming 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.

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

Reference Language

1. It is the working language of the committee.
2. It is the defining language.
3. The characters are determined by ease of mutual understanding and not by any computer limitations, coders notation, or pure mathematical notation.
4. It is the basic reference and guide for compiler builders.
5. It is the guide for all hardware representations.
6. It is the guide for transliterating from publication language to any locally appropriate hardware representations.
7. The main publications of the ALGOL language itself will use the reference representation.

Publication Language

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

Hardware Representations

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

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

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

DESCRIPTION OF THE REFERENCE LANGUAGE

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

Statements are supported by declarations which are not themselves computing instructions but inform the translator of the existence and certain properties of objects appearing in statements, such as the class of numbers taken on as values by a variable, the dimension of an

array of numbers, or even the set of rules defining a function. A sequence of declarations followed by a sequence of statements and enclosed between **begin** and **end** constitutes a block. Every declaration appears in a block in this way and is valid only for that block.

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

In the sequel the syntax and semantics of the language will be given.[4]

1.1. Formalism for Syntactic Description

The syntax will be described with the aid of metalinguistic formulae.[5] Their interpretation is best explained by an example

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

Sequences of characters enclosed in the brackets < > represent metalinguistic variables whose values are sequences of symbols. The marks ::= and | (the latter with the meaning of **or**) are metalinguistic connectives. Any mark in a formula, which is not a variable or a connective, denotes itself (or the class of marks which are similar to it). Juxtaposition of marks and/or variables in a formula signifies juxtaposition of the sequences denoted. Thus the formula above gives a recursive rule for the formation of values of the variable <ab>. It indicates that <ab> may have the value (or [or that given some legitimate value of <ab>, another may be formed by following it with the character (or by following it with some value of the variable <d>. If the values of <d> are the decimal digits, some values of <ab> are:

[(((1(37(
(12345(
(((
[86

In order to facilitate the study, the symbols used for distinguishing the metalinguistic variables (i.e. the sequences of characters appearing within the brackets < > as ab in the above example) have been chosen to be words describing approximately the nature of the corresponding variable. Where words which have appeared in this manner are used elsewhere in the text they will refer to the corresponding syntactic definition. In addition some formulae have been given in more than one place.

Definition:

<empty> ::=
(i.e. the null string of symbols).

[4] 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.

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

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[6] (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>
<arithmetic operator> ::= + | − | × | / | ÷ | ↑
<relational operator> ::= < | ≦ | = | ≧ | > | ≠
<logical operator> ::= ≡ | ⊃ | ∨ | ∧ | ¬
<sequential operator> ::= **go to**|**if**|**then**|**else**|**for**|**do**[7]
<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 ...

[6] It should be particularly noted that throughout the reference language underlining [in typewritten copy; boldface 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 including headings — Ed.], boldface will be used for no other purpose.

[7] **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.

a program the following "comment" conventions hold:

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

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

2.4. 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>
<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,,−‘[[[‘∧=/:’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 expression> | <Boolean expression> | <designational expression>

3.1. Variables

3.1.1. Syntax

⟨variable identifier⟩ ::= ⟨identifier⟩
⟨simple variable⟩ ::= ⟨variable identifier⟩
⟨subscript expression⟩ ::= ⟨arithmetic expression⟩
⟨subscript list⟩ ::= ⟨subscript expression⟩ | ⟨subscript list⟩, ⟨subscript expression⟩
⟨array identifier⟩ ::= ⟨identifier⟩
⟨subscripted variable⟩ ::= ⟨array identifier⟩ [⟨subscript list⟩]
⟨variable⟩ ::= ⟨simple variable⟩ | ⟨subscripted variable⟩

3.1.2. Examples

epsilon
detA
a17
Q[7,2]
x[*sin*(*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
$omega\times sum\uparrow cos(y+z\times 3)/7.394_{10}-8\uparrow w[i+2,8]\uparrow$
$\quad(a-3/y+vu\uparrow 8)$

Simple arithmetic expression:

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

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**
$\quad k\neq y$ **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 × have the conventional meaning (addition, subtraction, and multiplication). The type of the expression will be **integer** if both of the operands are of **integer** type, otherwise **real**.

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

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

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 ÷ 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⟩↑⟨primary⟩ denotes exponentiation, where the factor is the base and the primary is the exponent. Thus, for example,

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

while

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

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 $r\leq 0$, 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:   ↑
second:  × / ÷
third:   + −
```

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

3.3.6. Arithmetics of **real** quantities

Numbers and variables of type **real** must be interpreted in the sense of numerical analysis, i.e. as entities defined inherently with only a finite accuracy. Similarly, the possibility of the occurrence of a finite deviation from the mathematically defined result in any arithmetic expression is explicitly understood. No exact arithmetic will be specified, however, and it is indeed understood that different hardware representations may evaluate arithmetic expressions differently. The control of the possible consequences of such differences must be carried out by 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⟩ ::= < | ≤ | = | ≥ | > | ≠
⟨relation⟩ ::= ⟨simple arithmetic expression⟩
    ⟨relational operator⟩⟨simple arithmetic expression⟩
⟨Boolean primary⟩ ::= ⟨logical value⟩|⟨variable⟩|
    ⟨function designator⟩|⟨relation⟩|(⟨Boolean expression⟩)
⟨Boolean secondary⟩ ::= ⟨Boolean primary⟩|¬⟨Boolean primary⟩
⟨Boolean factor⟩ ::= ⟨Boolean secondary⟩|
    ⟨Boolean factor⟩∧⟨Boolean secondary⟩
⟨Boolean term⟩ ::= ⟨Boolean factor⟩|⟨Boolean term⟩
    ∨⟨Boolean factor⟩
⟨implication⟩ ::= ⟨Boolean term⟩|⟨implication⟩⊃⟨Boolean term⟩
⟨simple Boolean⟩ ::= ⟨implication⟩|
    ⟨simple Boolean⟩≡⟨implication⟩
⟨Boolean expression⟩ ::= ⟨simple Boolean⟩|
    ⟨if clause⟩⟨simple Boolean⟩ else ⟨Boolean expression⟩
```

3.4.2. Examples

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

3.4.3. Semantics

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

3.4.4. Types

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

3.4.5. The operators

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

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

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

3.4.6. Precedence of operators

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

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

```
first:   arithmetic expressions according to section 3.3.5.
second:  < ≤ = ≥ > ≠
third:   ¬
fourth:  ∧
fifth:   ∨
sixth:   ⊃
seventh: ≡
```

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

3.5. Designational Expressions

3.5.1. Syntax

```
⟨label⟩ ::= ⟨identifier⟩|⟨unsigned integer⟩
⟨switch identifier⟩ ::= ⟨identifier⟩
⟨switch designator⟩ ::= ⟨switch identifier⟩[⟨subscript expression⟩]
⟨simple designational expression⟩ ::= ⟨label⟩|⟨switch designator⟩|
    (⟨designational expression⟩)
⟨designational expression⟩ ::= ⟨simple designational expression⟩|
    ⟨if clause⟩⟨simple designational expression⟩ else
    ⟨designational expression⟩
```

3.5.2. Examples

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

3.5.3. Semantics

A designational expression is a rule for obtaining a label of a statement (cf. section 4. Statements). Again the principle of the evaluation is entirely analogous to that of arithmetic expressions (section 3.3.3). In the general case the Boolean expressions of the if clauses will select a simple designational expression. If this is a label the desired result is already found. A switch designator refers to the corresponding switch declaration (cf. section 5.3.

Switch Declarations) and by the actual numerical value of its subscript expression selects one of the designational expressions listed in the switch declaration by counting these from left to right. Since the designational expression thus selected may again be a switch designator this evaluation is obviously a recursive process.

3.5.4. The subscript expression

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

3.5.5. Unsigned integers as labels

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

4. Statements

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

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

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

4.1. Compound Statements and Blocks

4.1.1. Syntax

<unlabelled basic statement> ::= <assignment statement> | <go to statement> | <dummy statement> | <procedure statement>
<basic statement> ::= <unlabelled basic statement> | <label>: <basic statement>
<unconditional statement> ::= <basic statement> | <compound statement> | <block>
<statement> ::= <unconditional statement> | <conditional statement> | <for statement>
<compound tail> ::= <statement> **end** | <statement> ; <compound tail>
<block head> ::= **begin** <declaration> | <block head> ; <declaration>
<unlabelled compound> ::= **begin** <compound tail>
<unlabelled block> ::= <block head> ; <compound tail>
<compound statement> ::= <unlabelled compound> | <label>: <compound statement>
<block> ::= <unlabelled block> | <label>: <block>
<program> ::= <block> | <compound statement>

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

Compound statement:

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

Block:

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

It should be kept in mind that each of the 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 nonlocal to it, i.e. will represent the same entity inside the block and in the level immediately outside it. A label separated by a colon from a statement, i.e. labelling that statement, behaves as though declared in the head of the smallest embracing block, i.e. the smallest block whose brackets **begin** and **end** enclose that statement. In this context a procedure body must be considered as if it were enclosed by **begin** and **end** and treated as a block.

Since a statement of a block may again itself be a block the concepts local and nonlocal to a block must be understood recursively. Thus an identifier, which is nonlocal to a block A, may or may not be nonlocal to the block B in which A is one statement.

4.2. Assignment Statements

4.2.1. Syntax

<left part> ::= <variable> := | <procedure identifier> :=
<left part list> ::= <left part> | <left part list><left part>
<assignment statement> ::= <left part list><arithmetic expression> | <left part list><Boolean expression>

4.2.2. Examples

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

4.2.3. Semantics

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

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

4.2.3.2. The expression of the statement is evaluated.

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

4.2.4. Types

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

$$entier(E + 0.5)$$

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

4.3. 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 ∨ P≤Q then AA: begin if q<v then a := v/s
              else y := 2×a end
         else if v>s then a := v−q else if v>s−1
              then go to S
```

4.5.3. Semantics

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

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

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

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

and

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

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

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

ment 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

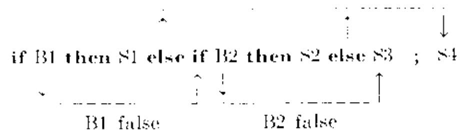
else ⟨unconditional statement⟩

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

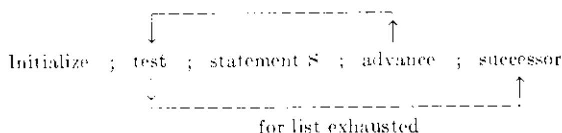
⟨for list element⟩ ::= ⟨arithmetic expression⟩|
⟨arithmetic expression⟩ **step** ⟨arithmetic expression⟩ **until** ⟨arithmetic expression⟩|⟨arithmetic expression⟩ **while** ⟨Boolean expression⟩
⟨for list⟩ ::= ⟨for list element⟩|⟨for list⟩, ⟨for list element⟩
⟨for clause⟩ ::= **for** ⟨variable⟩ := ⟨for 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, $V1\times 2$ **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** $(V-C)\times$ *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⟩|⟨letter string⟩⟨letter⟩

<parameter delimiter> ::= , |) <letter string> : (
<actual parameter list> ::= <actual parameter> |
  <actual parameter list> <parameter delimiter>
  <actual parameter>
<actual parameter part> ::= <empty> |
  (<actual parameter list>)
<procedure statement> ::= <procedure identifier>
  <actual parameter part>

4.7.2. Examples

Spur (*A*) Order: (7) Result to: (*V*)
Transpose (*W*, *v*+1)
Absmax (*A*, *N*, *M*, *Yy*, *I*, *K*)
Innerproduct (*A*[*t*, *P*, *u*], *B*[*P*], 10, *P*, *Y*)

These examples correspond to examples given in section 5.4.2.

4.7.3. Semantics

A procedure statement serves to invoke (call for) the execution of a procedure body (cf. section 5.4. Procedure Declarations). Where the procedure body is a statement written in ALGOL the effect of this execution will be equivalent to the effect of performing the following operations on the program at the time of execution of the procedure statement:

4.7.3.1. Value assignment (call by value)

All formal parameters quoted in the value part of the procedure declaration heading are assigned the values (cf. section 2.8. Values and Types) of the corresponding actual parameters, these assignments being considered as being performed explicitly before entering the procedure body. The effect is as though an additional block embracing the procedure body were created in which these assignments were made to variables local to this fictitious block with types as given in the corresponding specifications (cf section 5.4.5). As a consequence, variables called by value are to be considered as nonlocal to the body of the procedure, but local to the fictitious block (cf. section 5.4.3).

4.7.3.2. Name replacement (call by name)

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

4.7.3.3. Body replacement and execution

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

4.7.4. Actual-formal correspondence

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

4.7.5. Restrictions

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

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

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

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

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

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

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

4.7.6. Deleted.

4.7.7. Parameter delimiters

All parameter delimiters are understood to be equivalent. No correspondence between the parameter delimiters used in a procedure statement and those used in the procedure heading is expected beyond their number being the

same. Thus the information conveyed by using the elaborate ones is entirely optional.

4.7.8. Procedure body expressed in code

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

5. Declarations

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

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

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

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

Syntax.

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

5.1. Type Declarations

5.1.1. Syntax

⟨type list⟩ ::= ⟨simple variable⟩|
 ⟨simple variable⟩ , ⟨type list⟩
⟨type⟩ ::= **real** | **integer** | **Boolean**
⟨local or own type⟩ ::= ⟨type⟩|**own** ⟨type⟩
⟨type declaration⟩ ::= ⟨local or own type⟩⟨type list⟩

5.1.2. Examples

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

5.1.3. Semantics

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

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

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

5.2. Array Declarations

5.2.1. Syntax

⟨lower bound⟩ ::= ⟨arithmetic expression⟩
⟨upper bound⟩ ::= ⟨arithmetic expression⟩
⟨bound pair⟩ ::= ⟨lower bound⟩:⟨upper bound⟩
⟨bound pair list⟩ ::= ⟨bound pair⟩|⟨bound pair list⟩,⟨bound pair⟩
⟨array segment⟩ ::= ⟨array identifier⟩[⟨bound pair list⟩]|
 ⟨array identifier⟩,⟨array segment⟩
⟨array list⟩ ::= ⟨array segment⟩|⟨array list⟩,⟨array segment⟩
⟨array declaration⟩ ::= **array** ⟨array list⟩|⟨local or own type⟩
 array ⟨array list⟩

5.2.2. Examples

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

5.2.3. Semantics

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

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

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

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

5.2.4. Lower upper bound expressions

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

5.2.4.2. The expressions can only depend on variables and procedures which are nonlocal 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. How-

ever, even if an array is declared **own** the values of the corresponding subscripted variables will, at any time, be defined only for those of these variables which have subscripts within the most recently calculated subscript bounds.

5.3. Switch Declarations

5.3.1. Syntax

```
⟨switch list⟩ ::= ⟨designational expression⟩|
    ⟨switch list⟩,⟨designational expression⟩
⟨switch declaration⟩ ::= switch ⟨switch identifier⟩:=⟨switch list⟩
```

5.3.2. Examples

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

5.3.3. Semantics

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

5.3.4. Evaluation of expressions in the switch list

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

5.3.5. Influence of scopes

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

5.4. Procedure Declarations

5.4.1. Syntax

```
⟨formal parameter⟩ ::= ⟨identifier⟩
⟨formal parameter list⟩ ::= ⟨formal parameter⟩|
        ⟨formal parameter list⟩⟨parameter delimiter⟩
        ⟨formal parameter⟩
⟨formal parameter part⟩ ::= ⟨empty⟩|(⟨formal parameter list⟩)
⟨identifier list⟩ ::= ⟨identifier⟩|⟨identifier list⟩,⟨identifier⟩
⟨value part⟩ ::= value⟨identifier list⟩ ; |⟨empty⟩
⟨specifier⟩ ::= string|⟨type⟩|array|⟨type⟩array|label|switch|
    procedure|⟨type⟩procedure
⟨specification part⟩ ::= ⟨empty⟩|⟨specifier⟩⟨identifier list⟩ ; |
    ⟨specification part⟩⟨specifier⟩⟨identifier list⟩ ;
⟨procedure heading⟩ ::= ⟨procedure identifier⟩
    ⟨formal parameter part⟩ ; ⟨value part⟩⟨specification part⟩
⟨procedure body⟩ ::= ⟨statement⟩|⟨code⟩
⟨procedure declaration⟩ ::=
    procedure ⟨procedure heading⟩⟨procedure body⟩|
    ⟨type⟩ procedure ⟨procedure heading⟩⟨procedure body⟩
```

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

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

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

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

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

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

5.4.3. Semantics

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

block, whether it has the form of one or not. Consequently the scope of any label labelling a statement within the body or the body itself can never extend beyond the procedure body. In addition, if the identifier of a formal parameter is declared anew within the procedure body (including the case of its use as a label as in section 4.1.3), it is thereby given a local significance and actual parameters which correspond to it are inaccessible throughout the scope of this inner local quantity.

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

EXAMPLE 2.*

```
procedure RK(x,y,n,FKT,eps,eta,xE,yE,fi) ; value x,y ;
integer n ; Boolean fi ; real x,eps,eta,xE ; array
y,yE ; procedure FKT ;
comment: RK integrates the system y'k=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 un-
known functions yk(x). The order n of the system. The procedure
FKT(x,y,n,z) which represents the system to be integrated, i.e.
the set of functions fk. The tolerance values eps and eta which
govern the accuracy of the numerical integration. The end of the
integration interval xE. The output parameter yE which repre-
sents 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 mesh-
points x0, x1, ..., xn, then the procedure must be called repeat-
edly (with x=xk, xE=xk+1, for k=0, 1, ..., n-1) and then the
later calls may occur with fi=false which saves computing time.
The input parameters of FKT must be x,y,n, the output parameter
z represents the set of derivatives z[k]=fk(x,y[1], y[2], ..., y[n])
for x and the actual y's. A procedure comp enters as a nonlocal
identifier ;
begin
  array z,y1,y2,y3[1:n] ; real x1,x2,x3,H ; Boolean out ;
  integer k,j ; own real s,Hs ;
  procedure RK1ST(x,y,h,xe,ye) ; real x,h,xe ; array
        y,ye ;
    comment: RK1ST integrates one single RUNGE-KUTTA
        with initial values x,y[k] which yields the output
        parameters xe=x+h and ye[k], the latter being the
        solution at xe. Important: the parameters n, FKT, z
        enter RK1ST as nonlocal entities ;
    begin
      array w[1:n], a[1:5] ; integer k,j ;
      a[1] := a[2] := a[5] := h/2 ; a[3] := a[4] := h ;
      xe := x ;
      for k := 1 step 1 until n do ye[k] := w[k] := y[k] ;
      for j := 1 step 1 until 4 do
      begin
        FKT(xe,w,n,z) ;
        xe := x+a[j] ;
        for k := 1 step 1 until n do
        begin
          w[k] := y[k]+a[j]×z[k] ;
          ye[k] := ye[k] + a[j+1]×z[k]/3
```

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

```
      end k
     end j
   end RK1ST   ;
Begin of program:
      if fi then begin H := xE−x   ;   s := 0 end else H := Hs   ;
      out := false   ;
AA: if (x+2.01×H−xE>0) ≡ (H>0) then
      begin Hs := H   ;   out := true   ;   H := (xE−x)/2
        end if   ;
      RK1ST (x,y,2×H,x1,y1)   ;
BB: RK1ST (x,y,H,x2,y2)   ;   RK1ST(x2,y2,H,x3,y3)   ;
      for k := 1 step 1 until n do
            if comp(y1[k],y3[k],eta)>eps then go to CC   ;
      comment: comp(a,b,c) is a function designator, the value
        of which is the absolute value of the difference of the
        mantissae of a and b, after the exponents of these quan-
        tities have been made equal to the largest of the exponents
        of the originally given parameters a,b,c   ;
      x := x3   ;   if out then go to DD   ;
      for k := 1 step 1 until n do y[k] := y3[k]   ;
      if s=5 then begin s := 0   ;   H := 2×H end if   ;
      s := s+1   ;   go to AA   ;
CC: H := 0.5×H   ;   out := false   ;   x1 := x2   ;
      for k := 1 step 1 until n do y1[k] := y2[k]   ;
      go to BB   ;
DD: for k := 1 step 1 until n do yE[k] := y3[k]
end RK
```

ALPHABETIC INDEX OF DEFINITIONS OF CONCEPTS AND SYNTACTIC UNITS

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

def Following the abbreviation "def", reference to the syntactic definition (if any) is given.

synt Following the abbreviation "synt", references to the occurrences in metalinguistic formulae are given. References already quoted in the def-group are not repeated.

text Following the word "text", the references to definitions given in the text are given.

The basic symbols represented by signs other than underlined words [in typewritten copy; boldface in printed copy—Ed.] have been collected at the beginning.

The examples have been ignored in compiling the index.

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

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

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

⟨code⟩, synt 5.4.1 text 4.7.8, 5.4.6
colon :, synt 2.3, 3.2.1, 4.1.1, 4.5.1, 4.6.1, 4.7.1, 5.2.1
colon equal :=, synt 2.3, 4.2.1, 4.6.1, 5.3.1
comma ,, synt 2.3, 3.1.1, 3.2.1, 4.6.1, 4.7.1, 5.1.1, 5.2.1, 5.3.1, 5.4.1
comment, synt 2.3
comment convention, text 2.3
⟨compound statement⟩, def 4.1.1 synt 4.5.1 text 1
⟨compound tail⟩, def 4.1.1
⟨conditional statement⟩, def 4.5.1 synt 4.1.1 text 4.5.3

⟨decimal fraction⟩, def 2.5.1
⟨decimal number⟩, def 2.5.1 text 2.5.3
decimal point ., synt 2.3, 2.5.1
⟨declaration⟩, def 5 synt 4.1.1 text 1, 5 (complete section)
⟨declarator⟩, def 2.3
⟨delimiter⟩, def 2.3 synt 2
⟨designational expression⟩, def 3.5.1 synt 3, 4.3.1., 5.3.1 text 3.5.3
⟨digit⟩, def 2.2.1 synt 2, 2.1.1, 2.5.1
dimension, text 5.2.3.2
divide / ÷, synt 2.3, 3.3.1 text 3.3.4.2
do, synt 2.3, 4.6.1
⟨dummy statement⟩, def 4.4.1 synt 4.1.1 text 4.4.3

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

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

go to, synt 2.3, 4.3.1
⟨go to statement⟩, def 4.3.1 synt 4.1.1 text 4.3.3

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

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

minus −, synt 2.3, 2.5.1, 3.3.1 text 3.3.4.1
multiply ×, synt 2.3, 3.3.1 text 3.3.4.1
⟨multiplying operator⟩, def 3.3.1

nonlocal, text 4.1.3
⟨number⟩, def 2.5.1 text 2.5.3, 2.5.4

⟨open string⟩, def 2.6.1
⟨operator⟩, def 2.3
own, synt 2.3, 5.1.1 text 5, 5.2.5

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

quantity, text 2.7

real, synt 2.3, 5.1.1 text 5.1.3
⟨relation⟩, def 3.4.1 text 3.4.5
⟨relational operator⟩, def 2.3, 3.4.1

scope, text 2.7
semicolon ;, synt 2.3, 4.1.1, 5.4.1
⟨separator⟩, def 2.3
⟨sequential operator⟩, def 2.3
⟨simple arithmetic expression⟩, def 3.3.1 text 3.3.3
⟨simple Boolean⟩, def 3.4.1
⟨simple designational expression⟩, def 3.5.1
⟨simple variable⟩, def 3.1.1 synt 5.1.1 text 2.4.3
space ␣, synt 2.3 text 2.3, 2.6.3
⟨specification part⟩, def 5.4.1 text 5.4.5
⟨specificator⟩, def 2.3
⟨specifier⟩, def 5.4.1
standard function, text 3.2.4, 3.2.5
⟨statement⟩, def 4.1.1, synt 4.5.1, 4.6.1, 5.4.1 text 4 (complete section)
statement bracket, see: **begin end**
step, synt 2.3, 4.6.1 text 4.6.4.2
string, synt 2.3, 5.4.1
⟨string⟩, def 2.6.1 synt 3.2.1, 4.7.1 text 2.6.3
string quotes ‘ ’, synt 2.3, 2.6.1, text 2.6.3
subscript, text 3.1.4.1
subscript bound, text 5.2.3.1
subscript brackets [ ], synt 2.3, 3.1.1, 3.5.1, 5.2.1
⟨subscripted variable⟩, def 3.1.1 text 3.1.4.1
⟨subscript expression⟩, def 3.1.1 synt 3.5.1
⟨subscript list⟩, def 3.1.1
successor, text 4
switch, synt 2.3, 5.3.1, 5.4.1
⟨switch declaration⟩, def 5.3.1 synt 5 text 5.3.3
⟨switch designator⟩, def 3.5.1 text 3.5.3
⟨switch identifier⟩, def 3.5.1 synt 3.2.1, 4.7.1, 5.3.1
⟨switch list⟩, def 5.3.1

⟨term⟩, def 3.3.1
ten ₁₀, synt 2.3, 2.5.1
then, synt 2.3, 3.3.1, 4.5.1
transfer function, text 3.2.5
true, synt 2.2.2
⟨type⟩, def 5.1.1 synt 5.4.1 text 2.8
⟨type declaration⟩, def 5.1.1 synt 5 text 5.1.3
⟨type list⟩, def 5.1.1

⟨unconditional statement⟩, def 4.1 .1, 4.5.1
⟨unlabelled basic statement⟩, def 4.1.1
⟨unlabelled block⟩, def 4.1.1
⟨unlabelled compound⟩, def 4.1.1
⟨unsigned integer⟩, def 2.5.1, 3.5.1
⟨unsigned number⟩, def 2.5.1 synt 3.3.1
until, synt 2.3, 4.6.1 text 4.6.4.2
⟨upper bound⟩, def 5.2.1 text 5.2.4

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

while, synt 2.3, 4.6.1 text 4.6.4.3

END OF THE REPORT

AL.7a.18

CHAPTER 7b

Features of ALGOL-60 which are changed in ALGOL-20

This section lists those aspects of the Report which do not hold for ALGOL-20. The section numbers refer to the Report and page numbers refer to this manual.

2.1 Only upper case letters are available.

2.2.2,2.3 The basic symbols indicated by underlined identifiers in the reference language (boldface in the report) are replaced in Algol-20 by identifiers with the same spelling. These identifiers, which may not be used for any other purpose, are referred to as "reserved identifiers". In addition, certain other reserved identifiers have been added to Algol-20. See Chapter 2, page 3ff.

2.3 A change to a new line of input text has the same significance as a blank space (except that strings may not continue beyond the end of a line). See Chapter 6a.

2.3 The characters ⊃ and ÷ are not available. The characters ≡, ×, ≤ and ≥ are available in different forms. See Chapter 2, page 2.

2.4 Identifiers may not contain spaces. However, see Chapter 2, page 5, for alternative punctuation.

2.5 See Chapter 2, page 6 for the range of values of meaningful numbers.

2.6.1 Since ALGOL-20 cannot distinguish between a left and right string quote, strings may not contain strings.

3.1.4.2 Array subscripts are truncated, not rounded, when they are evaluated.

3.3.4.2 ÷ is not available. See Chapter 2, page 2.

3.3.4.3 ↑ produces a value of type real when it is applied to any combination of real and integer values.

3.4 ⊃ is not available. See Chapter 2, page 8.

3.4.6.1 ≡ has the same precedence as =.

3.5 A label may not be an unsigned integer.

4.1 ALGOL-20 defines <program>::=<unlabeled block>|<unlabeled compound>. Thus the first character of a program must be a begin.

4.1.3 If the first occurrence of a label in the block in which it is defined is as an actual parameter, it is necessary to declare it as a label in that block head.

4.6 The controlled variable in a for statement may only be a simple variable.

4.7.3.1 Arrays cannot be called by value.

5.2 Dynamic own arrays are not allowed.

5.4 Recursive procedures are not available.

5.4.5 All formal parameters must be specified.

CHAPTER 7c

Restrictions on ALGOL-20
to transform it into a subset of ALGOL-60

The user of ALGOL-20 may use many abilities which are not part of ALGOL-60, since the translator at Carnegie Tech implements an extension of the language. If a program (or procedure) is to be sent outside of Carnegie Tech, however, the programmer may wish to restrict himself to those aspects of our system which are part of the standard language. To do so, he must obey the rules given in this section.

Anything which gives a note (except for notes 1 and 2) indicates a deviation of ALGOL-20 from ALGOL-60 and so should not be used.

All left parts in a statement must be of the same type.

Boolean variables must not occur as primaries in arithmetic expressions.

Arithmetic variables must not occur as primaries in Boolean expressions.

Go must be followed by to.

Nothing may be assumed about the initial value of a variable - including own variables.

"." is not a legal character in identifiers.

Constants may not end with a decimal point.

Variables may only be of type real, integer or Boolean.

The value of a for variable is undefined after the for statement has run to completion.

If a unary operator follows another operator, it and its operand must usually be surrounded by parentheses.

None of the following exist in ALGOL-60 and must not be used:

octal constants

alphanumeric string constants

step...while for-list elements

privileged identifiers (with their privileged meanings)

label declarations

nested substitutions

the operators ↓ and ←

the reserved words max, min, and mod

library procedures

SY statements

CO statements

WHAT

the operator = used to mean ≡

input/output

|| comment convention

THIS PUBLICATION IS AVAILABLE AT:

THE BOOK STORE
BAKER HALL
CARNEGIE INSTITUTE OF TECHNOLOGY
PITTSBURGH, PENNSYLVANIA 15213

THE SALE PRICE ($1.25)* REFLECTS THE COST
OF PRINTING AND DISTRIBUTION ONLY.

*(Add $.06 sales tax for purchase in Pa.)
*(Add $.20 postage for mail orders)