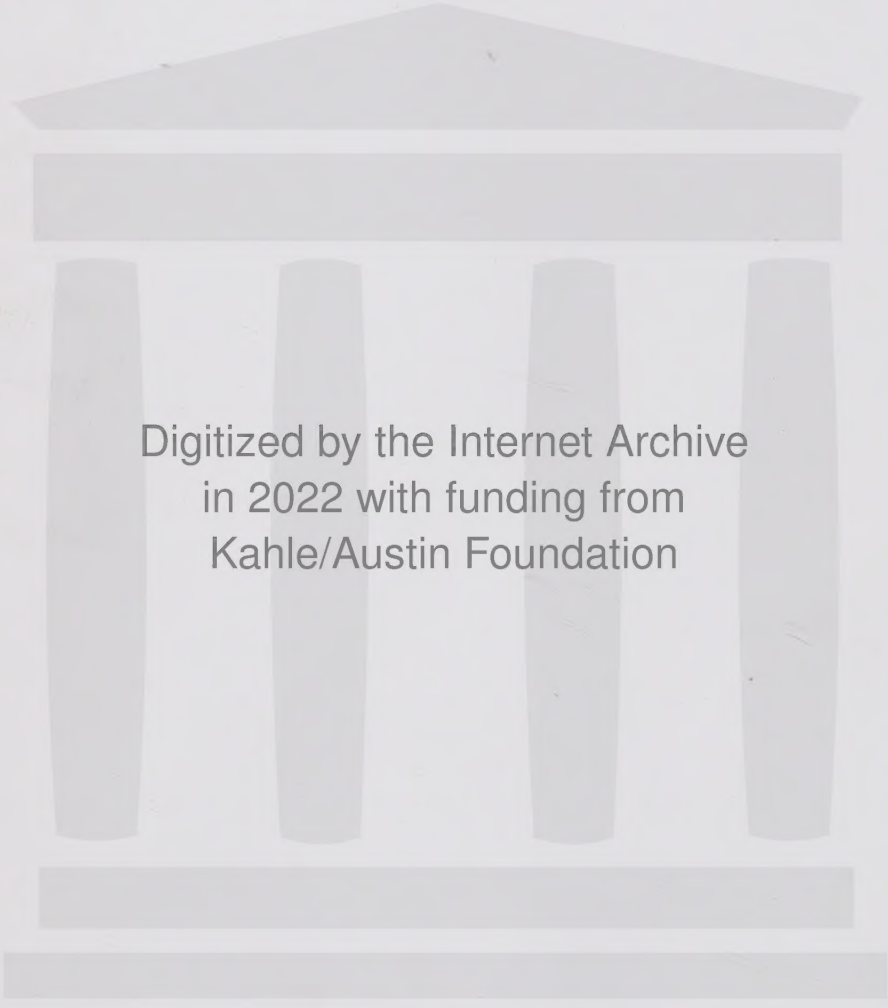
AN INTRODUCTION TO BALGOL

OAKFORD & GERE



AN INTRODUCTION TO BALGOL

ROBERT V. OAKFORD & JAMES M. GERE

School of Engineering, Stanford University

WADSWORTH PUBLISHING COMPANY, INC., BELMONT, CALIFORNIA

PRINTED IN THE UNITED STATES OF AMERICA.

L. C. Cat. Card. No.: 61-18824

PREFACE

The purpose of this book is to introduce the reader to a language that can be used for describing computational processes. The language is BALGOL, the algorithmic language used in communicating with the Burroughs 220 digital computer. BALGOL is very similar to ALGOL, the "universal" computer language.

The approach which is followed is to introduce the reader to the language in a step-by-step manner, beginning with very simple concepts and gradually expanding to include all of the important features of the language. The introduction of new concepts is brought about in many cases by means of illustrative examples. The semantics and syntax of the language are discussed only to the extent necessary to enable the reader to write programs. In other words, this is intended to be an introductory approach by which the reader can learn to use the language. For more detailed definitions of the semantics and syntax than is given here, the reader should refer to the reference manual titled Burroughs Algebraic Compiler (published by the Burroughs Corporation).

The illustrative examples in the text do not involve sophisticated mathematics; rather, they have been chosen so that the reader will have no difficulty in understanding the problem to be solved, and can therefore devote his full attention to the BALGOL description of the computational process.

Obviously, there are many different ways in which the problems in the illustrative examples can be solved. Our objective has been to present programs which illustrate certain features of the BALGOL language, and the reader is urged to follow the illustrative programs with this thought in mind. Subsequently, the reader may wish to develop for himself one or more alternative programs which solve the same problem.

Questions and problems appear at the ends of many of the articles. The reader is urged to program solutions to the problems in order to develop a working familiarity with the language.

A brief description of the Burroughs 220 computer is given in the first chapter. However, it should be realized that a person can write programs in BALGOL without any specific knowledge of how the computer operates. The second chapter introduces the language and most readers will find that the material presented here is sufficient for the majority of their programing purposes. The third chapter contains some generalizations and summaries of the topics which have been discussed in Chapter 2.

This book was developed originally as a set of lecture notes for a one-week computer seminar given at Stanford University. The seminar was sponsored by the School of Engineering, in cooperation with the Computation Center, for the benefit of members of the engineering faculty. The authors are indebted to Dr. Joseph M. Pettit, Dean of the School of Engineering, for his interest and support during the preparation of the notes and for permission to expand them into an introductory text.

The authors are indebted also to Professors George Forsythe and John Herriot, Directors of the Stanford Computation Center, and to members of their staff, for helpful advice and for providing an opportunity to use the notes in several BALGOL classes. The authors also wish to express their appreciation to (Mrs.) Marjorie Horsley, (Mrs.) Doris Blanton, and (Mrs.) Jane Solnar, who typed the lecture notes, and to Miss Rosemarie Stampfel who typed the final manuscript.

R. V. Oakford

J. M. Gere

CONTENTS

CHAPTER 1

INTRODUCTION

1.1 The Burroughs 220 Computer

The Burroughs 220 is a digital computer which can be programed to perform automatically a computational process. This is accomplished by providing the computer with a program, consisting of a sequence of instructions. Normally, each instruction specifies one or more operations to be executed by the computer and identifies the operands, or quantities which are to be operated on. The program can be stored within the computer itself, thus permitting successive instructions to be interpreted and executed automatically by the computer without human intervention. Hence, the Burroughs 220 computer is said to be of the stored-program type.

The computer has five basic components which are identified as follows: input unit, memory, control unit, processing unit, and output unit. A schematic diagram illustrating the relationship of these units is shown in Figure 1. The input unit performs the function of supplying the computer with information. The information may be instructions or data, and it may be available on punched cards, magnetic tape, or punched paper tape. The input unit translates the information into the language of the computer and sends it to the memory unit for storage, as indicated by arrow A in Figure 1.

The memory unit is a place where information may be recorded for future reference. Information is stored in words of fixed size. Each word consists of a sign (plus or minus) followed by 10 decimal digits, each digit being 0, 1, 2, ... , 8, or 9. All information, whether instructions or data, is stored in memory in words of this type.

Information flows to memory from the input unit, as described above, and also from the processing unit which performs operations on the data. The processing unit must receive data from memory and the results of the operations performed will normally be stored in memory. Hence, there is a two-way flow of information between these units (arrows B and C). Among the operations that can be performed in the processing unit are arithmetic operations (such as addition, multiplication, etc.), logical operations (such as "less than", "equal to", etc.), and data-transmission operations for the purpose of relocating information stored in memory.

The control unit locates successive instructions, analyzes them, and causes the specified operations to be performed on the specified data. The control unit will normally obtain instructions directly from memory (arrow D). However, an important feature of any automatic computer is that instructions in the program may be modified during execution of the program.

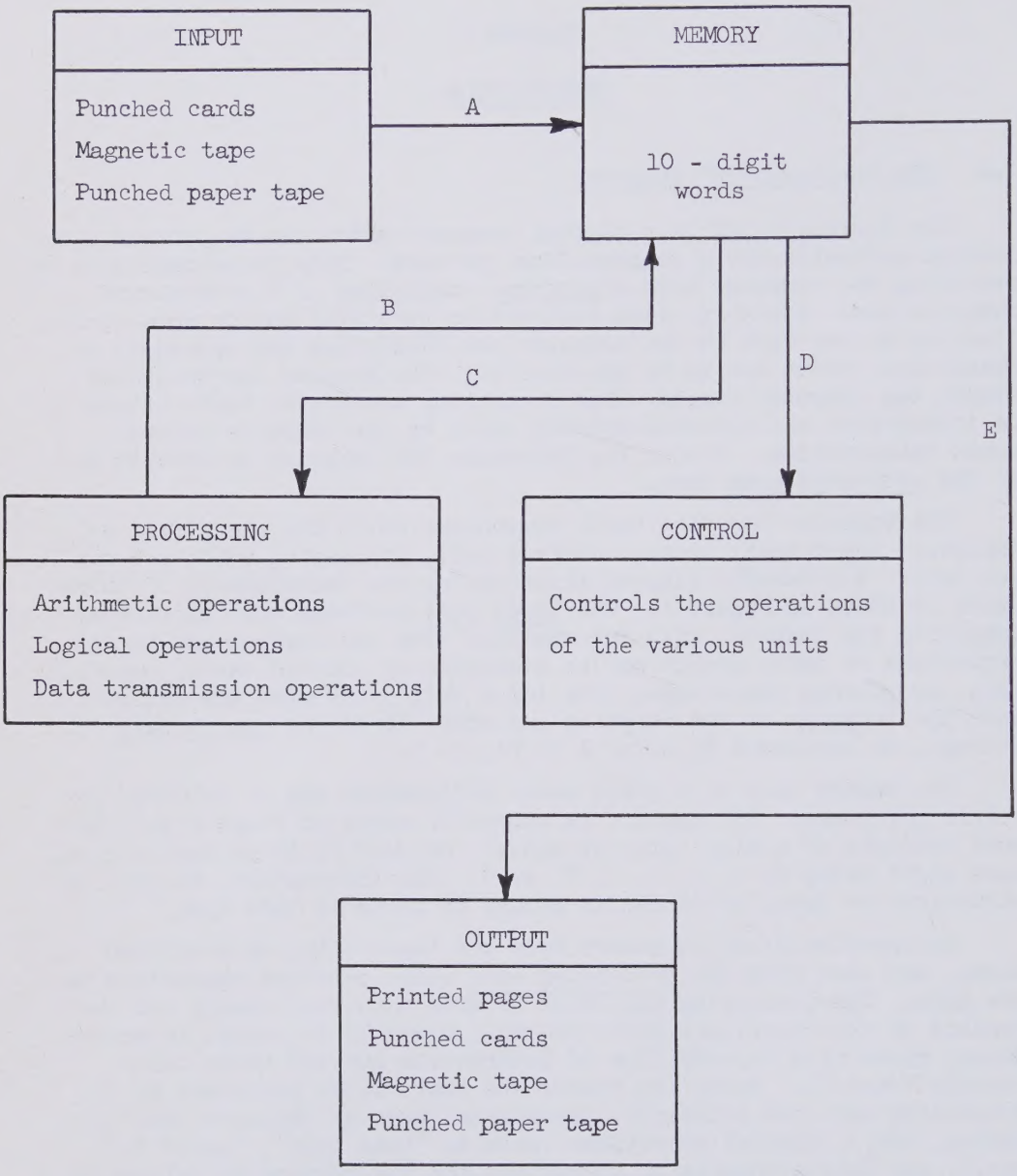


Fig. 1. Schematic Diagram of Burroughs 220 Computer

The output unit translates information from memory (arrow E) to an appropriate form specified by the programer, and then transfers it to the output medium. The medium may be printed pages, punched cards, magnetic tape, or punched paper tape.

In addition to the computer itself, a computation facility will normally include auxiliary equipment used in the preparation of the information that is to be processed by the computer. A key-punch is similar in function to a typewriter except that it is used to translate alphabetic or numeric information into punched-card form. A card sorter is used for sorting punched cards in some desired manner by detecting punches contained in the cards. A reproducing punch is used for copying information from one set of punched cards to another. An interpreter is a machine which prints on a punched card (for convenience in reading the card) the information which is punched in the card. A lister prints on paper the information which is punched in cards. Other pieces of equipment are available, but those listed above are the most essential for ordinary computing work.

1.2 Machine Languages

The term "machine language" refers to the language which is inherent in the construction of the computer itself. Since each word in memory in the Burroughs 220 consists of a 10-digit number (with sign), it is logical that machine-language instructions should consist of 10-digit numbers. Hence, it follows also that a machine-language program consists of a sequence of such numbers. For example, the sequence of instructions

```
0000101525
0000191526
```

has the following significance to the computer: set the contents of word 1525 into the accumulator (a part of the processing unit), and then add the current contents of the accumulator directly to the contents of word 1526. It is apparent from this simple example that machine-language programing requires that the programer write one or more machine-language instructions for each arithmetic operation in a sequence of calculations. Furthermore, it is necessary for him to know at all times the locations in memory (that is, the word locations) where data are stored.

It is obvious from the preceeding discussion that machine-language programing has many undesirable features. It is not only tedious and time consuming for the programer, but it is an unnatural language. Hence, the potential for human error is great and the isolation of an error is frequently quite difficult. Furthermore, a program can be executed only by the particular type of computer for which it is written. It is also quite difficult for a person to read a machine-language program and determine therefrom the computational process described by the program. To overcome these difficulties, algorithmic languages have been developed.

1.3 Algorithmic Languages

An algorithmic language is a language for describing computational processes. It does so by means of instructions which are very similar to those which would be used in writing English and in writing mathematical relationships. Hence algorithmic languages are easy to learn and the meaning of the instructions is readily recognized.*

ALGOL is one such algorithmic language. It was developed by a joint effort of a group of European and American scientists and was described in the COMMUNICATIONS OF THE ACM (Vol. 3, No. 5, May 1960, pp. 299-314). It is hoped by many people that this language will become the "universal" language for describing computational processes.

A language of this type is machine independent, but a computer may be programed to translate an ALGOL language program into a machine-language program. Such a translator is referred to as an ALGOL compiler. Thus a program written in ALGOL can be executed on any computer that has an ALGOL compiler.

There are numerous other algorithmic languages in existence today, for example, FORTRAN, FORTRANSIT, MAD, BALGOL, etc., all of which have many features in common. However, each such language requires a different compiler and hence they are not interchangeable for computational purposes. They are sufficiently alike, however, that once a person becomes familiar with one of these languages, it is relatively easy for him to become familiar with another.

From this point on, our discussion will be devoted exclusively to the BALGOL language, which may be considered as a Burroughs 220 dialect of ALGOL. The Burroughs 220 Algebraic Compiler (hereafter called either BALCOM or the compiler for the sake of brevity) for the BALGOL language was developed by employees of the Burroughs Corporation and is a proprietary document. The basic input medium for BALCOM** is punched cards and output is by means of either cards or printed pages.

---

* For the origin of the word algorithm, we take the following quote from Elementary Number Theory, by J. V. Uspensky and M. A. Heaslet, McGraw-Hill Book Company, Inc., 1939, p. 26: "The word algorithm, like several other mathematical expressions, comes from the Arabic, being a corruption of Al Khowarizmi, the name of an Arabian mathematician of the ninth century, whose writings were prominent in bringing the present method of numeration to the Occident. During the middle ages the word algorithm referred simply to the use of Hindu-Arabic numerals, but at present it applies to any formalized procedure whereby requested mathematical objects are found by a definite chain of operations, each operation requiring the results of preceding ones."

** The designation BAC-220 is widely used in place of BALCOM.

CHAPTER 2

PROGRAMING IN BALGOL

2.1 An Elementary BALGOL Program

A program in BALGOL is written by the programer on standard Coding Forms. All alphabetic characters are written in upper case letters, except the letter O which is written with a slash $\emptyset$ in order to distinguish it from zero. All numeric digits are written in the conventional manner. Other characters, such as +, -, ., /, (,), are also used with well-defined meanings which will be described later. The semicolon ; and the dollar sign $ are interchangeable and are used as a separator between statements, clauses, etc.

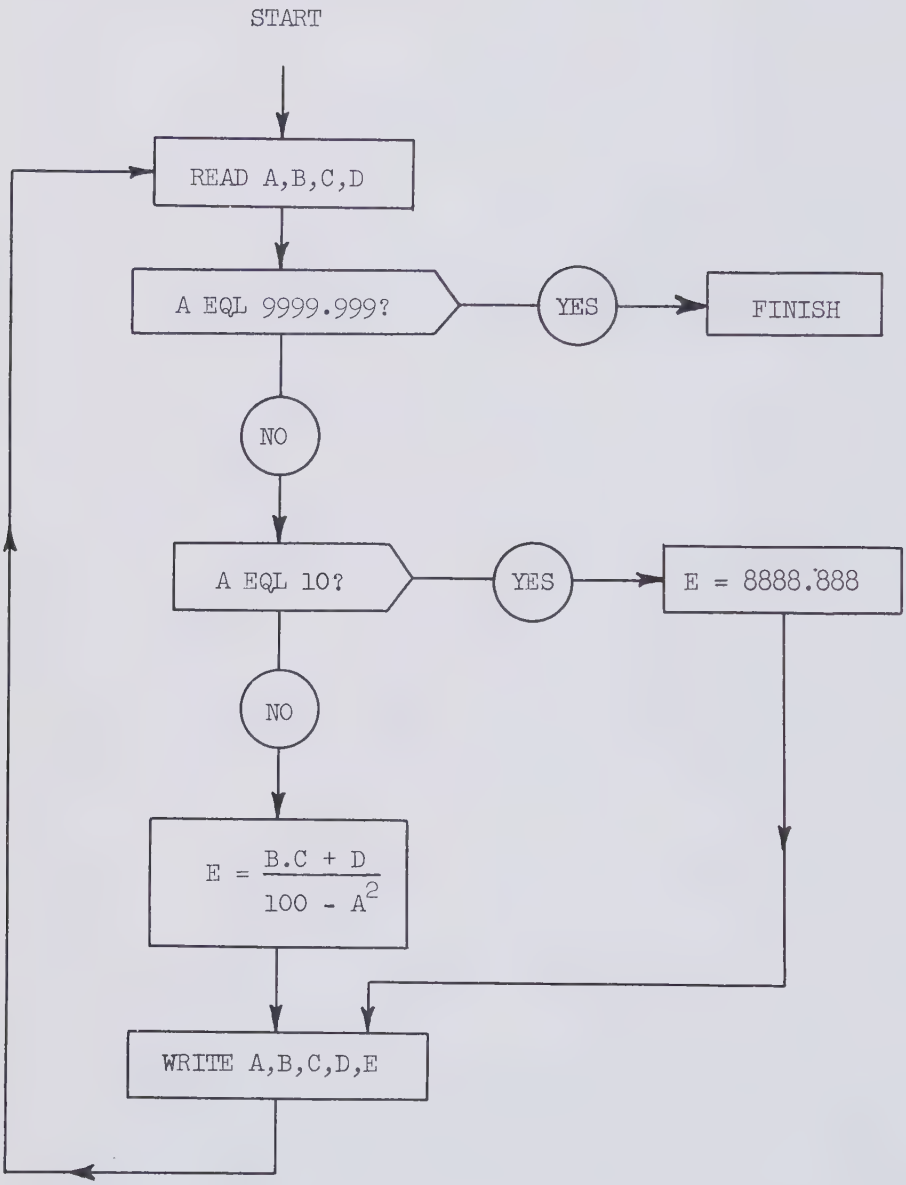
Each line on the Coding Form will be punched in a single card. In order to distinguish them from other types of cards, program cards must contain a 2 in column 1. The program itself may occupy columns 2 to 72, while columns 73 to 80 are reserved for identification purposes, such as sequential numbering of the cards. Data cards are distinguished by a 5 in column 1, with the remainder of the card (columns 2 to 80) available for data.

A program for the following problem illustrates some of the fundamental concepts of the BALGOL language. In addition, the program illustrates the steps followed by the programer in preparing his program and running it on the computer.

Statement of Problem: Let us assume that a set of data cards is available, each card containing four numbers which will be identified as A, B, C, and D. It is desired to calculate $E = (B \cdot C + D)/(100 - A^2)$ for each set of values of A, B, C, and D. The results are to be printed on paper with the given values of A, B, C, and D and the corresponding calculated value of E appearing on one line. Provisions are to be made for identifying the last data card to the computer and for indicating that E is undefined whenever the value A = 10 is encountered. To accomplish the first provision, an extra data card will be appended containing the value 9999.999 for A (it is known that this value of A will not appear elsewhere in the data cards) and, to accomplish the second provision, E will be set equal to 8888.888 whenever A equals 10. The numbers 9999.999 and 8888.888 have been selected arbitrarily.

The flow chart on the following page outlines the sequence of operations to be performed in executing this problem.

Flow Chart:


START
READ A,B,C,D
A EQL 9999.999?
YES
FINISH
NO
A EQL 10?
YES
E = 8888.888
NO
E = B.C + D / 100 - A^2
WRITE A,B,C,D,E

The following BALGOL program describes the computational process illustrated in the flow chart. Note that the 2 required in column 1 of the program cards and the 5 required in column 1 of the data cards are omitted for the sake of clarity. The cards in the program are numbered sequentially for convenient reference in the explanation which follows the program. These numbers may be omitted in the actual program if desired.

A detailed explanation of each part of the BALGOL program is given in the discussion following the program.

BALGOL Program	Card No.
Job Card	1
Lodx Card	2
L1.. READ (;; IDS1);	3
INPUT IDS1 (A,B,C,D);	4
IF A EQL 9999.999; GØ TØ L3;	5
IF A EQL 10; BEGIN E = 8888.888; GØ TØ L2 END;	6
E = (B.C + D)/(100 - A*2);	7
L2.. WRITE (;; ØDS1, FØRM1);	8
ØUTPUT ØDS1 (A,B,C,D,E);	9
FØRMAT FØRM1 (5X11.3, W);	10
GØ TØ L1;	11
L3.. FINISH;	12

Typical Data Cards

1.5	1.64	72.439	3.0	13
-12.0	2.34	57.194	7.0	14
10.0	1.91	87.992	-7.0	15
3.5	2.02	-78.700	6.0	16
3.0	2.0	90.0	2.0	17
9999.999	1.0	1.0	1.0	18

After the program is written on the BALGOL Coding Form (see Fig. 2 which shows the preceding program as written on a typical form), it is punched on cards in the indicated sequence. Then the programer assembles the deck of cards for processing by the computer. The assembled deck consists of the header cards, the program cards, and the data cards (in that order). The assembled deck is placed in the hopper of the card-read input unit of the Burroughs 220 by the computer operator. Successive cards are read automatically into the computer under control of BALCOM. The compiler translates the BALGOL program into machine-language instructions and the resulting program is stored in memory. This phase of the operation is known as the "compile phase" and is terminated by the FINISH declaration (which must always appear as the last statement in a program). The program itself is listed during the compile phase (see the upper part of Fig. 3, which shows a typical listing of the preceding program). Many of the possible syntactical errors which can be made by the programer are detected by the compiler during this phase and appropriate error messages are interspersed in the listing. This feature is of considerable aid in removing errors from programs (a process usually referred to as "debugging"). Of course, not all errors will be detected by BALCOM.

When compilation is completed, the "execution phase" begins automatically and the first statement in the program is executed. The remaining statements in the program are executed consecutively (except as modified by GO TO or conditional statements in the program itself) until the program is completed. The results are printed on paper (or punched on cards), as shown in the lower part of Fig. 3. Note that supplementary information may also be printed, depending upon the particular compiler which is being used. For example, the listing shown in Fig. 3 includes information pertaining to the time on and off the computer, the last of the word locations (0279) in memory occupied by a machine-language instruction, the first of the word locations (9260) in memory occupied by the variables in the program (the intervening memory locations are unused), and a message which indicates that all data cards were read.

Throughout this book, any word which is part of the BALGOL language is underlined in the illustrative programs. This makes it simple to distinguish words which are part of the language (called "reserved words") from those which the programer has defined. Of course, when writing an actual program no underlines are used. A complete listing of all reserved words is given in Art. 3.12.

BALGOL CODING FORM

PROBLEM: ILLUST. PROB. - ART. 2.1

CODER: JOHN DOE — DATE: 11/10/61 — CHARGE NO: A-011 — PAGE 1 OF 1

1	STATEMENTS/DATA (2–72)	SEQ.NO./ID (73–80)
2	A-011 JØB 11/10/61 1 MIN. JØHN DØE (ILLUST. PRØB. ART. 2.1)	
2	LØDX BALGØL	
2	L1.. READ($$ IDS1) $	
2	INPUT IDS1(A, B, C, D) $	3
2	IF A EQL 9999.999 $ GØ TØ L3 $	4
2	IF A EQL 10 $ BEGIN E = 8888.888 $ GØ TØ L2 END $	5
2	E = (B.C + D)/(100 - A*2) $	6
2	L2.. WRITE($$ ØDS1, FØRM1) $	7
2	ØUTPUT ØDS1(A, B, C, D, E) $	8
2	FØRMAT FØRM1(5X11.3, W) $	9
2	GØ TØ L1 $	10
2	L3.. FINISH $	11
		12
5	1.5 1.64 72.439 3.0	
5	-12.0 2.34 57.194 7.0	
5	10.0 1.91 87.992 -7.0	
5	3.5 2.02 -78.700 6.0	
5	3.0 2.0 90.0 2.0	
5	9999.999 1.0 1.0 1.0	

Column markers: 1 2 5 8 11 14 17 20 23 26 29 32 35 41 51 61 71 | 73 80

Necessary:
1. BALGOL statements: 2 in col. 1. Data: 5 in col. 1.
2. Statements may be continued on one or more lines, in which case col. 2 immediately follows col. 72.
3. Align indentations to right of dotted vertical lines.
4. Place a slash on letter O (Ø) to distinguish from zero (0).
5. Use upper case letter I to distinguish from number 1.
6. Never erase; strike out the line.

Recommended:
1. Place statement labels (if any) in cols. 2 - 7.
2. Place one or at most two statements on a single line.
3. To facilitate reading, indent statements according to level in the program.
4. Start associated statement brackets BEGIN, END in same column and indent bracketed statements at least three more spaces.

FIG. 2 Program as Written on Typical BALGOL Coding Form

```
$$$  TIME ON.   1358
 A-011          JOB  11/10/61  1 MIN    JOHN DOE (ILLUST. PROB. ART. 2.1)
                         BURROUGHS ALGEBRAIC COMPILER
                    L1..  READ($$ IDS1) $                                    3
                    INPUT IDS1(A,B,C,D) $                                    4
                    IF A EQL 9999.999 $ GO TO L3 $                           5
                    IF A EQL 10 $ BEGIN E = 8888.888 $ GO TO L2 END $        6
                    E = (B.C + D)/(100 - A*2) $                              7
                    L2..  WRITE($$ ODS1, FORM1) $                            8
                    OUTPUT ODS1(A,B,C,D,E) $                                 9
                    FORMAT FORM1(5X11.3, W) $                               10
                    GO TO L1 $                                              11
                    L3..  FINISH $                                          12
COMPILED PROGRAM ENDS AT 0279
PROGRAM VARIABLES BEGIN AT 9260


      1.500       1.640      72.439       3.000       1.246
    -12.000       2.340      57.194       7.000      -3.200
     10.000       1.910      87.992      -7.000    8888.888
      3.500       2.020     -78.700       6.000      -1.743
      3.000       2.000      90.000       2.000       2.000

   NO CARDS NOT READ

$$$  TIME OFF   1359
```

FIG. 3 Listing of Results of Compilation and Execution of Program

Comments on Program:

Card 1. The Job Card is required at the beginning of every program. A typical format is as follows:

Columns	Entry
1	2
2-7	(Charge Number)
16-18	JØB
21-29	(Date)
30-36	(Estimated running time in minutes)
40-80	(Identifying information, such as name of programer, phone number, etc.)

Card 2. The Load-Execute Card (or Lodx Card) must be the second card in every program. A typical format is as follows:

Columns	Entry
1	2
16-19	LØDX
21-26	BALGØL

The Job Card and the Load-Execute Card are frequently called "header cards" and serve several functions. The Job Card provides information needed both by the computer operator and by the accounting staff of the Computation Center, and also identifies the author of the program. The Load-Execute Card identifies the program as a BALGOL program, so that the appropriate compiler (BALCOM) will be called into service.

Card 3. READ Statement. A statement defines an operation to be performed by the computer. In this case, execution of the READ statement causes only one data card to be read. However, in general, card reading will continue automatically until a value has been assigned to each variable in the referenced input-data set. The READ statement refers to the input-data set by an identifier, in this case IDS1. For the present, we will consider that the READ statement always consists of the word READ followed by a left parenthesis, two semicolons, the identifier of an input-data set, and a right parenthesis. The explanation for the presence of the two semicolons is given at the end of Art. 2.12 (see p. 64).

Identifiers are names chosen by the programer to identify various entities in the program; e.g., variables, input-data sets, output-data sets, editorial formats, etc. An identifier must start with an alphabetic character (that is, a letter); otherwise it can be an almost arbitrary combination of alphabetic characters and numeric characters (that is, digits). However, each identifier must be unique, must not contain more than 50 characters, and must not conflict with the list of reserved words given in Art. 3.12 (p. 87). No special characters, such as commas, periods, plus signs, etc., nor spaces, can be contained in an identifier.

A label is a name used to identify a statement for reference purposes. A label may be either an identifier or an integer (in the case of an integer, leading zeros are ignored; thus, the labels 0027, 027, and 27 are equivalent). The distinguishing characteristic of a label is the separator .. that follows it and separates it from the statement it labels. In general, the function of separators in BALGOL is analogous to that of punctuation marks in the English language. In this example the READ statement is labeled L1 so that the statement may be referred to later on in the program.

All statements are separated from one another by a semicolon (or dollar sign).

Card 4. INPUT Declaration. The purpose of a declaration is to define the properties of an identifier used elsewhere in the program. It is not an execution statement. The declaration always begins with a declarator, in this case the word INPUT. An INPUT declaration defines one or more input-data sets, each of which is given a name in the form of an identifier. In this example, there is one input-data set which is arbitrarily given the name IDS1. The input-data set consists of the list of variables A, B, C, D to which numerical values will be assigned when the identifier IDS1 is referred to by a READ statement (Card 3). Successive values from a data card (or from data cards) will be assigned to successive variables in the input-data set. The programer is responsible for synchronizing the sequence of values on data cards with the sequence of variables in the input-data set.

If two or more input-data sets are to be defined in a single INPUT declaration, then commas are used to separate the sets. For example, the following declaration would define three input-data sets:

```
INPUT IDS1(X1,Y1,Z1), IDS2(X2,Y2,Z2), IDS3(X3,Y3,Z3)
```

All declarations are separated from each other and from statements by a semicolon (or dollar sign).

Card 5. IF Statement. The IF statement is a conditional statement which defines an operation to be performed only if a specified condition is satisfied. The IF statement has two parts: first, an IF clause which specifies the condition, and second, a statement (which is actually a statement within the IF statement), which is executed only if the condition is satisfied.

In this example the IF statement specifies that if A equals 9999.999, the statement GØ TØ L3 is to be executed. If A is not equal to 9999.999, the GØ TØ L3 statement is not executed and in this example the following statement (Card 6) will be executed.

The phrase A EQL 9999.999 is called a relation. Among the relations that can be specified in the IF clause are the following:

BALGOL	Mathematical Equivalent
E_1 LSS E_2	$E_1 < E_2$
E_1 LEQ E_2	$E_1 \leq E_2$
E_1 EQL E_2	$E_1 = E_2$
E_1 NEQ E_2	$E_1 \neq E_2$
E_1 GEQ E_2	$E_1 \geq E_2$
E_1 GTR E_2	$E_1 > E_2$

In the above relations E_1 and E_2 are arithmetic expressions (see discussion of Card 7 below). The operators LSS, LEQ, etc., are called relational operators and must be preceded and followed by a space. It is possible to specify more than one arithmetic relation in an IF clause by constructing a Boolean expression, as illustrated in the next article.

The GO TO statement is used to branch to a labeled statement, thereby controlling the sequence of statement execution. The words GO TO are followed by the label of the statement which is to be executed next. Note that the symbol .. following a label (e.g., L3..) acts as a separator that distinguishes a label from an identifier and is not considered to be part of the label. Thus the separator .. does not appear after the label in the GO TO statement itself.

Card 6. IF Statement Containing a Compound Statement. For syntactical purposes it is frequently desirable that a sequence of statements shall be regarded as a single statement. This is accomplished by enclosing the sequence of statements within a pair of "brackets" in the form

$$\text{BEGIN } S_1 \text{ ; } S_2 \text{ ; } S_3 \text{ ... ; } S_n \text{ END}$$

Such a statement is called a compound statement. The symbols S_1, S_2, ... S_n each indicate either a simple or a compound statement. (Thus a compound statement may be nested within a compound statement.) The symbols BEGIN and END serve as enclosing "brackets" for the compound statement and may be replaced by left and right parentheses if desired. (A semicolon may be inserted between the last statement S_n and END if desired.)

In this example the compound statement BEGIN E = 8888.888; GØ TØ L2 END is executed whenever the condition A EQL 10 is satisfied; otherwise the compound statement is not executed and in this example the program proceeds to the next statement (Card 7).

If the condition A EQL 10 is satisfied, the value 8888.888 is arbitrarily assigned to E. The symbol = in BALGOL does not have the usual algebraic meaning of equality, but instead it has the meaning assign. Hence, the relational operator EQL and the assignment symbol = are definitely not interchangeable. The distinction between these quantities will become clearer as the symbol = is used subsequently. After E has been assigned the value 8888.888, the GØ TØ L2 statement is executed.

Card 7. Assignment Statement. An assignment statement is expressed in general as:

$$V = E$$

or

variable = expression

In other words, the assignment symbol = means that the current value of the expression on its right-hand side is to be assigned to the variable on its left.

In this example the arithmetic expression on the right must be evaluated for the values currently assigned to the variables A, B, C, D. The result is assigned to the variable E.

The concept of a variable in BALGOL is the same as in algebra, namely, it is an identifier that represents a quantity that may assume any one of a series of values.

The concept of an arithmetic expression in BALGOL is also the same as in algebra. Furthermore, the symbols and syntax are almost identical. The permissible arithmetic operators are:

Arithmetic Operator in BALGOL	Meaning
*	exponentiation
.	multiplication
/	division
+, -	addition, subtraction

Exponentiation (A*B) is undefined when A = 0 and B $\leq$ 0 or when A < 0 and B is nonintegral. Otherwise, A and B may be positive or negative, and integer or noninteger. Division (A/B) is undefined when B = 0.

The elements of an arithmetic expression are numbers, variables, arithmetic operators, and parentheses. Within parentheses the order of precedence for performing the arithmetic operations is exponentiation, multiplication, division, and addition or subtraction. As in algebra, parentheses should be used to remove ambiguities. For example, A/B/C should be either (A/B)/C or else A/(B/C). Similarly, A*B*C should be either A*(B*C) or (A*B)*C.

Omission of the multiplication operator . is permitted whenever this will not result in ambiguity. For example, the product of 2X + 7 and 3Y + 8 can be written in the form

$$(2X + 7)(3Y + 8)$$

which is equivalent to

$$(2X + 7).(3Y + 8)$$

Similarly, the product of 4 and X can be written in the form 4X, which is equivalent to either 4.X or 4(X). When multiplying constants, parentheses must be used to eliminate ambiguities; thus 3 times 6 should be written as 3(6) or (3)6, rather than 3.6.

For convenient reference, the rules pertaining to arithmetic expressions are summarized in Art. 3.1 (p. 69).

Card 8. WRITE Statement. A WRITE statement refers first to an output-data set by its identifier (e.g., ØDS1), and secondly to a format specification by its identifier (e.g., FØRM1). The execution of the WRITE statement causes the values currently assigned to the variables in the referenced output-data set (ØDS1) to be printed (or punched) as specified in the referenced format specification (FØRM1). As in the case of the READ statement, two semicolons are required in the WRITE statement, for reasons which are explained later (see p. 64).

Card 9. OUTPUT Declaration. The purpose of an OUTPUT declaration is to define one or more output-data sets, each of which is given an arbitrarily selected name. In this example the data set consists of the variables A, B, C, D, E, and has been given the name ØDS1. The values currently assigned to the variables will be printed (or punched) when the identifier ØDS1 is referred to in a WRITE statement, provided that an appropriate format (FØRM1) is also referred to in the WRITE statement.

Card 10. FORMAT Declaration. The FORMAT declaration defines one or more format specifications (hereafter called formats) and specifies identifiers for the formats. The declaration consists of the declarator FORMAT followed by the list of formats each of which is preceded by an identifier. Each format is enclosed within parentheses. A format specifies the form of presentation of individual values in the output-data set, the spacing of successive values on the printed page (or on cards), and the output medium (printer or card punch). A format may also be used to define headings in alphabetic form. Because of the complexities of possible formats, only the format used in the example will be described at present. Generalizations and variations will be presented in later examples.

The format in this example consists of two parts, an editing phrase (5X11.3) and an activation phrase (W). There must be a suitable correspondence between editing phrases in the format and variables in the corresponding output-data set. In this case there are five variables (A, B, C, D, E) in the output-data set. The significance of the 5 in the editing phrase 5X11.3 is that the phrase is to be repeated five times. In other words, 5X11.3 is equivalent to writing X11.3, X11.3, X11.3, X11.3, X11.3, and thus it is seen that there is one editing phrase associated sequentially with each variable in the output-data set. The X indicates that the value currently assigned to the variable is to be written as a decimal number; the 11 indicates that the number is to be written in a field 11 columns wide; and the 3 indicates that 3 digits to the right of the decimal point will be printed (the remaining digits to the right are discarded). The number will be justified right in the field and the sign, if negative, will be printed to the left of the number. Thus the printed line will have the following appearance:

A	B	C	D	E
±XXX.XXX	±XXX.XXX	±XXX.XXX	±XXX.XXX	± XXX.XXX

The activation phrase W specifies that the results are to be printed with single spacing between lines.

Thus it is seen that the execution of a WRITE statement causes the sequence of values specified in the output-data set to be assembled as specified by the editing phrases in the format and printed or punched as specified by the activation phrase in the format.

Card 11. This GO TO statement specifies that the next statement to be executed is the one prefixed by the label L1. Therefore, as long as data cards are available, another data card will be read and the sequence of operations will begin anew.

Card 12. FINISH Statement. The FINISH statement indicates to the compiler the completion of the compile phase. Consequently the last statement in a program must be a FINISH statement, and furthermore the FINISH statement must not appear elsewhere in the program. After compilation is completed, the first statement in the program (in this example, Card 3) will be executed, thus beginning the execution phase.

Execution of the FINISH statement at the end of the execute phase indicates completion of program execution and hence card reading is automatically initiated. In this way the compilation of the next program in the input unit is begun automatically and the computer is not delayed, as is the case when the operator must start each program manually.

The reader will note that on page 8 reference was made to the FINISH declaration, whereas reference is made here to the FINISH statement. This dual reference results from the different roles of the word FINISH in the compile phase and the execute phase.

Cards 13 to 18. Typical Data Cards. Column 1 of each data card must contain the digit 5. This distinguishes a data card from other types of cards, such as program cards which have a 2 in column 1. Columns 2 to 80 of a data card may be used to store successive values. At least one blank column must separate successive values on the card. Furthermore, each value must be wholly contained on a single card. Successive values from a card are assigned to successive variables in the input-data set as they are encountered reading from left to right across the card.

In this example four values (A, B, C, D) are punched on each card, hence one card will be read upon each execution of the READ statement. The numbers on the data cards in this example must be written in decimal form; integer numbers will be discussed in Art. 2.3 (p. 24).

The use of the additional data card containing A = 9999.999 is redundant in this example and could be omitted if desired. If this data card were omitted, and if Card 5 in the program were omitted also, no difficulty would be encountered and the results of execution of the program would be the same. After printing the results of the last calculation for the variable E (see Card 8), the GO TO statement on Card 11 is executed. Then the READ statement (Card 3) is executed but, since there are no more data cards, the card which is read will be the first card in the next program. This card will be identified automatically and compilation of the next program will begin automatically, just as if the FINISH statement were executed.

Editorial Use of Blank Spaces and Line Advances. In general, the rules for the editorial use of blank spaces and the starting of a new line in the BALGOL program are intuitive by analogy to the English language. In the English language a blank space is used to indicate the end of a word or a number. Ambiguity may result if a blank space is inserted within a word or if a space is not inserted to separate adjacent words.

Similarly, in the BALGOL language a blank space may be used when necessary to indicate the end of a "word", e.g., an identifier, a reserved word, or a declarator. Ambiguity will result from the insertion of a blank space such that a single "word" is divided into two "words" or from omission of a blank space such that two "words" are combined into a single "word".

In many places in a program spaces are optional and the programer may suit his own convenience. For example, spaces are optional before and after arithmetic operators (A + B and A+B are equivalent), before and after the assignment symbol (A = 2 and A=2 are equivalent), before and after separators, before and after commas which separate items in a list (as in INPUT, OUTPUT, and FORMAT declarations), and before and after parentheses.

An advance from one line (or card) in the program to the next line is not the same as in English writing. There is no counterpart in BALGOL to the use of a hyphen in English to indicate a word that is separated between two successive lines. Furthermore the compiler does not recognize an advance from one program card to the next. Instead, column 72 of one program card is followed immediately by column 2 of the next card. Thus, if a word is broken at column 72 of one card and continued at column 2 of the next card, it will appear to BALCOM as a single word. Alternatively, if one word ends at column 72 of one card and the next word starts at column 2 of the next card, the two words will appear to BALCOM as a single word.

Subject to these general restrictions and to a very limited number of other restrictions (e.g., spaces are required to the left and the right of relational operators), spaces and line advances may be used freely by the programer to enhance the readability of his program.

One or more statements or declarations may appear on a single line, and a single statement or declaration may occupy more than one line.

PROBLEMS

2.1-1. (a) Which of the following may be used as a label? (b) Which may be used as an identifier? (c) Which are reserved words?

A729	ABCDEFG	001	2DELTA
BEFØRE	FINISH	AB+BA	12.5
1000	1	BESSEL	ØAKFØRD

2.1-2. Assume that A, B, and THETA are identifiers representing simple variables. Which of the following are syntactically correct as arithmetic expressions for

(a) the product of 7 and A:

7A A7 A.7 7.A

(b) the product of A and B:

AB BA A.B A(B)

(c) the product of 2.5 and A:

2.5A A2.5 A.2.5 A(2.5)

(d) the product of A and THETA:

A.THETA THETA.A ATHETA THETA(A)

(e) the product of 2.5 and 3.5:

(2.5)(3.5) 2.5(3.5) 2.5.3.5 2.53.5

2.1-3. (a) Why are compound statements needed? (b) In the following compound statement, is there any ambiguity in the last arithmetic expression? (c) Write a single assignment statement for Z (as defined in the following statement) in terms of numbers and the variable A only.

```
BEGIN X = (2.5A) + 4.0; Y = X*4; Z = X(Y + X/2) END
```

2.1-4. Prepare an arbitrary number of data cards, each containing one number (in decimal form, e.g., 2.756). It is desired to calculate the squares of the numbers on the cards. Write a BALGOL program which provides for reading the numbers from the cards (one at a time), squaring the numbers, and writing each number and its square on a single line. Thus, the pages of results will contain two columns with the original numbers in the first column and the corresponding squares in the second column.

2.1-5. Write a BALGOL program to calculate the cross-sectional area A and polar moment of inertia J for circular tubes of various sizes. Prepare a number of data cards that contain the outside and inside diameters (D2 and D1) for various sizes of tubes. (The numbers on the data cards should be in decimal form.) Read the diameters from the cards, calculate A and J, and list the results in four columns (D2, D1, A, J).

2.1-6. Prepare a number of data cards, each containing two (decimal) values A and B. It is desired to calculate values of the variables Y and Z defined as:

$$Y = (A + B)^3/3.5$$

$$Z = (A^3 - 8A^2B + 3AB^2 - B^3)/3.5$$

Write a program in BALGOL to read exactly five sets of values of A and B from cards (assuming that the number of data cards that follows the program is greater than five) and to calculate the corresponding values of Y and Z. List results in four columns with corresponding values of A, B, Y, and Z appearing in each row.

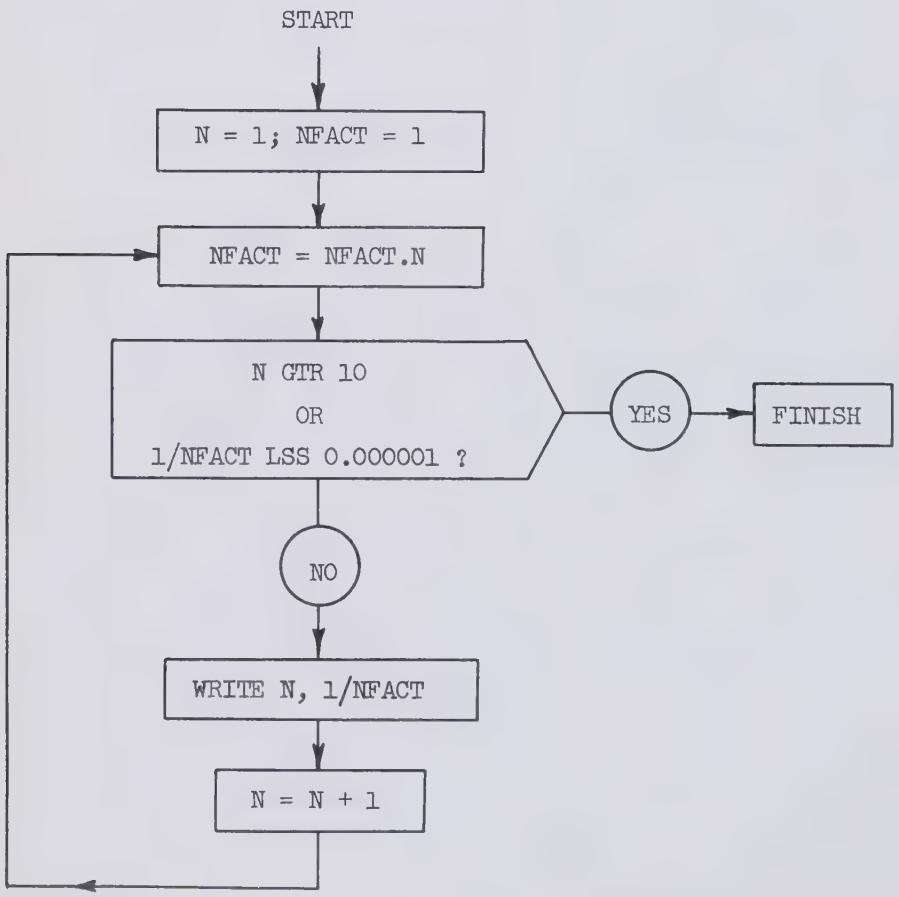
2.2 A Second Illustrative Program

The following problem illustrates some variations of the concepts discussed in the preceding article, as well as some additional features of the language.

Let us assume that we wish to compute a table of reciprocal factorials 1/N! for values of N equal to 1, 2, 3, ... The results are to be printed in two columns with the value of N in the first column and the corresponding value of 1/N! in the second column. The calculations are to be terminated when N becomes greater than 10 or when the value of the reciprocal factorial is less than 0.000001, whichever occurs first.

The following flow chart indicates the sequence of calculations to be defined in the BALGOL program. Note that in this example it is not necessary to read numbers from data cards for use in the program. Instead, all required numbers are either specified as constants or else calculated in the program.

Flow Chart:



The following BALGOL program describes the calculations shown in the flow chart.

BALGOL Program	Card No.
Job Card	1
Lodx Card	2
N = 1; NFACT = 1;	3
LA.. NFACT = NFACT.N;	4
IF (N GTR 10) ØR (1/NFACT LSS 0.000001);	5
GØ TØ LB;	6
WRITE (;; RESULT, FØRM);	7
ØUTPUT RESULT (N, 1/NFACT);	8
FØRMAT FØRM (B20, X5.1, B5, X10.8, W2);	9
N = N + 1; GØ TØ LA;	10
LB.. FINISH;	11

Comments on Program:

Card 3. The two statements on this card serve the purpose of assigning initially the value 1 to the variables N and NFACT, the latter being the identifier for the variable N factorial.

Card 4. This assignment statement assigns to the variable NFACT a value which is equal to the current value of NFACT times the current value of N. (The expression "current value" refers to the value currently assigned to the variable.) During the first cycle of execution of the program, the values of NFACT and N are each equal to 1, hence the "new" value assigned to NFACT is also 1. During the second cycle of execution (after N has been assigned the value 2; see Card 10), the execution of this statement will assign to NFACT the value 2, which is equal to the current (or "old") value of NFACT times the current value of N. During the third cycle, NFACT will be assigned the value 6, etc. Thus this single assignment statement serves to calculate the value of N! for each value of N.

Alternative forms for writing the product NFACT.N in this statement are NFACT(N) and (NFACT)N.

Cards 5 and 6. These two cards contain an IF statement consisting of an IF clause (Card 5) and a statement (Card 6). The clause contains a Boolean expression which is recognized by the presence of the Boolean operator OR. The expression contains two arithmetic relations, each contained in parentheses. In this case, if either one of the arithmetic relations is satisfied (or is "true") then the Boolean expression itself is said to be "true", and hence the IF clause is satisfied. Therefore, the statement immediately following the clause (and contained within the IF statement) is executed. If neither one of the two relations is satisfied, then the Boolean expression is said to be "false", the IF clause is not satisfied, and the statement following it is not executed. Instead, in this example, the next statement (Card 7) is executed.

If it is desired that the cycle of calculations in this program be repeated until both of the arithmetic relations are satisfied, then the Boolean operator AND would be used in place of OR. In other words, the Boolean expression

(N GTR 10) AND (1/NFACT LSS 0.000001)

is true only if both relations are satisfied.

Other Boolean operators are NOT, IMPL, and EQIV. Their meanings are discussed in Art. 3.2 (p. 71).

An extension of the IF statement is the EITHER IF statement (or the "alternative statement"), which is described in Art. 3.4 (p. 75).

Card 7. The execution of this statement results in the output-data set identified as RESULT being printed according to the format identified as FØRM.

Card 8. This declaration specifies that the output-data set RESULT consists of the values of N and 1/NFACT. Note that an expression (e.g., 1/NFACT) as well as a variable (e.g., N) may be used in an output-data set. The expression may have any degree of complexity. This is in contrast to input-data sets which must contain variables only, for the obvious reason that only variables can have values assigned to them.

Card 9. This declaration specifies the format for printing the output-data set. The editing phrase B20 specifies that there will be 20 blank spaces. The phrase X5.1, which is associated with N, specifies that N is to be printed in a field 5 columns wide with 1 digit to the right of the decimal point. The next phrase B5 specifies 5 blank spaces. The phrase X10.8 specifies that the value of 1/NFACT is to be printed in a field 10 columns wide with 8 digits to the right of the decimal point. Finally, the activation phrase W2 specifies printing of the preceding items with a single space both before and after printing (hence double spacing). A summary of the various editing and activation phrases is given in Art. 3.8 (p. 83).

It should be noted that by changing a few cards in this program, it becomes a program for listing factorials and their reciprocals for values of N covering any desired range. Also, by making use of numbers of type integer (discussed in the next article) some slight improvement in the format can be made.

PROBLEMS

2.2-1. Prepare a BALGOL program for calculating by direct addition the sum of the numbers 0.1, 0.2, 0.3, ... , 10.0 and writing the result. (Answer: SUM = 505.0)

2.2-2. Write a BALGOL program for calculating by direct addition the sum of the numbers 0.1, 0.2, 0.3, ... Terminate the calculations at the number N such that the sum is less than or equal to 100, but the addition of the next number (N + 0.1) would produce a sum greater than 100. The results to be printed consist of the number N and the corresponding sum. (Answer: N = 4.4; SUM = 99.0)

2.2-3. Prepare an arbitrary number of data cards each containing one number (in decimal form, e.g., 2.756). It is desired to calculate the sum of all the numbers. Write a BALGOL program which provides for reading the numbers from the cards, writing the numbers in a column, and then writing the sum of the numbers on a new line but not in the same column.

2.2-4. Write a program in BALGOL to calculate a table of values of the dynamic magnification factor MF equal to

$$\frac{1}{1 - r^2}$$

where r is the ratio of the frequency of the forcing function to the natural frequency of vibration of the dynamic system. Write the results in two columns, the first giving the value of r and the second giving the corresponding value of MF. Select values of r from 0 to 3 in intervals of 0.2, and provide for the fact that MF becomes infinite when r = 1.

2.2-5. Write a BALGOL program to compute the value of e by using the series

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \cdots$$

Terminate the series when the number of terms exceeds 200 or when the first term not included is less than 0.00001. Write the calculated value of e and the number of terms used in the calculations.

2.2-6. Write a BALGOL program to compute the sine of $\pi/6$ from the series

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

Terminate the calculations when the number of terms exceeds 50 or when the first term not included is less than 0.000001. Write the calculated value of the sine and the number of terms used to calculate it.

2.3 Numbers and Type Declarations

Numbers which are punched on data cards or which are incorporated as part of a BALGOL program may be represented in one of three different forms. These will be referred to as integer form, decimal form, and floating-point form.

A number represented in integer form will consist of a string of digits, for example, 17294. The maximum number of digits permitted is 10, since each word in the memory of the computer contains 10 digit positions. A plus or minus sign may precede the number. Leading zeros are ignored; thus, 001234 is treated as if it were 1234. An integer number is stored in memory in the same form in which it is written as data or as part of a program, except that leading zeros are supplied to fill up the word in memory. Some examples of numbers in integer form are as follows:

BALGOL Program	Data Card	Computer Memory
1234	1234	0000001234
-1234	-1234	-0000001234
-1234512345	-1234512345	-1234512345
123456123456	123456123456	Undefined
0	0	0000000000

A number represented in decimal form will consist of a string of digits containing a decimal point (e.g., 12.345). When the number appears as part of a BALGOL program it is necessary that the decimal point be imbedded between two digits (for example, 0.5 and 11.0 are acceptable forms, but .5 and 11. are not acceptable). On the other hand, the decimal point need not be imbedded when a number appears in a data card; thus, .5 and 11. are acceptable decimal forms on a data card. A plus or minus sign may precede a number in decimal form.

A number which appears in decimal form will automatically be stored in the computer memory in excess-fifty floating-point form. This notation is introduced in order to increase the range of the numeric values that can be represented in a 10-digit word, as pointed out later. To understand the excess-fifty floating-point form, consider a number containing no more than eight significant digits that is expressed in the form

$$\pm.\text{mmmmmmmm} \times 10^{cc-50}$$

where each m represents a significant digit in the number and where

.mmmmmmmmm $\geq$ 0.1. Each c also represents a digit so that cc-50 (called a scale factor) is a two-digit number such as 21, -17, 02, or -03. The excess-fifty floating-point form of the above number is

$$\pm \text{ ccmmmmmmmmm}$$

The number cc, called the characteristic, is equal to the scale factor (the power of 10 by which the number is multiplied) plus 50. The reason for choosing this convention for representing the scale factor is so that cc is always positive, thus obviating the need for two signs in a single word (one for the number and one for the power of 10). Thus, the characteristic cc is used to define the location of the decimal point in the actual number while the remaining eight digits (called the mantissa) are the significant digits in the actual number.

Some examples of numbers in excess-fifty floating-point form follow:

Number	Excess-fifty Floating-point Form
1	5110000000
10	5210000000
-1,234,567,800,000,000	-6612345678
0.0001	4710000000
-0.00000012345678	-4412345678

It is apparent that the largest value which is permitted for the number cc is 99 and the smallest value is 00. This means that the largest positive number which can be represented by the excess-fifty convention is 9999999999 which is equivalent to $0.99999999 \times 10^{49}$. Thus any number equal to or larger than 10^{49} is undefined by the excess-fifty convention. The smallest positive number which can be represented is 0010000000 or 0.1×10^{-50}. The excess-fifty convention assigns the value zero to any positive number smaller than this. Corresponding comments can be made concerning negative numbers.

In the absence of a floating-point scale factor convention, such as the excess-fifty convention, the ratio of the largest positive number to the smallest positive number (not equal to zero) that can be represented in a 10-digit word is 9999999999. For example, if the decimal point is implied to the left of the fourth digit position (counting from the right) the largest number that can be represented is 999999.9999 and the smallest nonzero number is 0.0001. Thus it is apparent that the use of the excess-fifty floating-point convention extends considerably the range of numbers which can be represented.

From the preceding discussion it is apparent that a decimal number can have no more than eight significant digits, inasmuch as it is stored in memory in floating-point form. Leading zeros are not counted as significant digits but trailing zeros are counted as significant digits.

Some examples of decimal numbers follow:

BALGOL Program	Data Cards	Computer Memory
123.0	123. or 123.0	5312300000
-123.45	-123.45	-5312345000
0.00012	.00012 or 0.00012	4712000000
12345678.9	12345678.9	Undefined
1234567800.0	1234567800.	Undefined
0.000012345678	.000012345678	4612345678
0.0 or 0	0. or 0	0000000000

The third way in which a number may be represented is the floating-point form, which is analogous to the familiar convention in which a number is multiplied by a power-of-ten scale factor (e.g.,1.5×10^6). Such a number may appear in a BALGOL program as a decimal number (or an integer number) followed by the scale factor. The scale factor is preceded on program cards by a double asterisk, which serves to identify the scale-factor convention to the compiler. In contrast, on data cards the same number must be represented as a decimal number followed by a scale factor which is preceded by a comma. The comma performs the same functions on data cards as the double asterisk used on the program cards. A floating-point number is always stored in memory in excess-fifty floating-point form, hence no more than eight significant digits may be used. Some examples are:

BALGOL Program	Data Cards	Computer Memory
1.23**2	1.23,2	5312300000
12.3**+1	12.3,+1	5312300000
-0.12345**3	-.12345,3	-5312345000
12**-5	12.,-5	4712000000
12345678**2	12345678.,2	6012345678
12345678**-12	12345678.,-12	4612345678
0	0	0000000000
0.12**75	.12,75	Undefined
0.12**-60	.12,-60	Undefined

In the last example the characteristic cc would have to be -10, and in the next to last example it would have to be 125. Both of these values are outside the permitted range of cc and hence the resulting numbers in memory are undefined.

It is seen from the above discussion that equivalent numbers in both decimal and floating-point form will have the same representation in computer memory, namely excess-fifty floating-point form.

Because all numbers of the kind discussed in this article are stored in memory in either integer or excess-fifty form, it is only necessary to distinguish between these two cases as far as arithmetic operations are concerned. Hence in the BALGOL language, numeric variables are said to be of integer type or floating-point type.*

However, when referring to numbers as they appear in the BALGOL program or on data cards it is necessary to distinguish between the three forms previously discussed: integer, decimal, and floating-point.**

Since integer arithmetic and floating-point arithmetic are not the same, it is necessary to declare in a BALGOL program the type of any variable. Then the computer will carry out the appropriate type of arithmetic when performing arithmetic operations with that variable.

The type of the variable represented by an identifier is specified by a type declaration. For example, FLOATING A, B, C declares that A, B, and C are of type floating-point (or for brevity, type floating). Similarly, INTEGER D, E declares that D and E are of type integer. Another possible type declaration is as follows: INTEGER OTHERWISE, which declares that all numeric variables not explicitly declared as type floating shall be considered as integers. Similarly, FLOATING OTHERWISE declares that all variables not otherwise declared as type integer shall be considered as type floating. However, this latter declaration is redundant since any variable not explicitly declared as to type is declared implicitly (or declared by default) to be of type floating. Thus, if all quantities which appear in a program are to be treated as floating, no type declaration is necessary.

It should be noted that in a type declaration the reserved word REAL is synonymous and interchangeable with the word FLOATING. (This is one of the few redundancies in the language.)

To show some of the differences between integer and floating-point arithmetic, consider the following illustrations:

---

* Boolean variables are discussed in Art. 3.2 (p. 71).

** In the reference manual Burroughs Algebraic Compiler (published by the Burroughs Corporation), these three forms are called integer, floating-point, and floating-point with a scale factor, respectively.

Example No. 1. (12,345,678)(1000) = 12,345,678,000
In floating-point arithmetic the result is 12,345,678,000 and is represented in memory as 6112345678. In integer arithmetic the result is 2345678000 since only 10 digits can be stored in memory and the number is truncated from the left.

Example No. 2. (11,111,111)(222) = 2,466,666,642
In floating-point the result is 2,466,666,600 (represented as 6024666666). In integer arithmetic the result is 2466666642.

Example No. 3. 15/6 = 2.5
In floating point the result is 2.5 (represented as 5125000000). In integer arithmetic the result is 2, since all digits to the right of the decimal point are dropped.

Example No. 4. 2/3 = 0.6666...
In floating-point the result is 0.66666666 (note that the result is not rounded) and is represented in memory as 5066666666. In integer arithmetic the result is 0.

Example No. 5. 12345678/10 = 1234567.8
In floating-point the result is 1234567.8 (represented as 5712345678). In integer arithmetic the result is 1234567. Note that this provides a means of isolating certain digits of a number, in this case the first seven digits (from the left). The first four digits of an eight digit number can be obtained by dividing by 10000 in integer arithmetic; the first two digits by dividing by 1000000, etc.

Example No. 6. 5,000,000,001 + 5,000,000,001 = 10,000,000,002
In integer arithmetic the result of this calculation is 2.

Finally, it should be observed that the input and output procedures do not take into account type declarations. In the case of input, the type of a number is determined solely by the form in which it appears in the data card. In other words, if the number is punched in the card in either decimal or floating-point form, it will be stored in memory in excess-fifty floating-point form; if punched in integer form, it will be stored in integer form.* It is the responsibility of the programer to ensure that the forms of the values on data cards are consistent with the declared types of the variables to which such values will be assigned.

In the case of output, the form in which the number is to be printed (or punched) is specified by the editing phrase. The output procedure assumes that the type of the value as stored in memory is consistent with the form specified in the editing phrase.

---

* The type of a number which appears explicitly in a BALGOL program is also determined by the form in which the number is written.

2.4 Illustrative Program

The following example illustrates the use of type declarations in distinguishing between numbers of integer and floating-point types, and also incorporates some additional features of the language.

Assume that it is desired to calculate a table of values of natural logairthms by using the series

$$\text{LN } X = 2\left[\frac{X-1}{X+1} + \frac{1}{3}\left(\frac{X-1}{X+1}\right)^3 + \frac{1}{5}\left(\frac{X-1}{X+1}\right)^5 + \cdots\right]$$

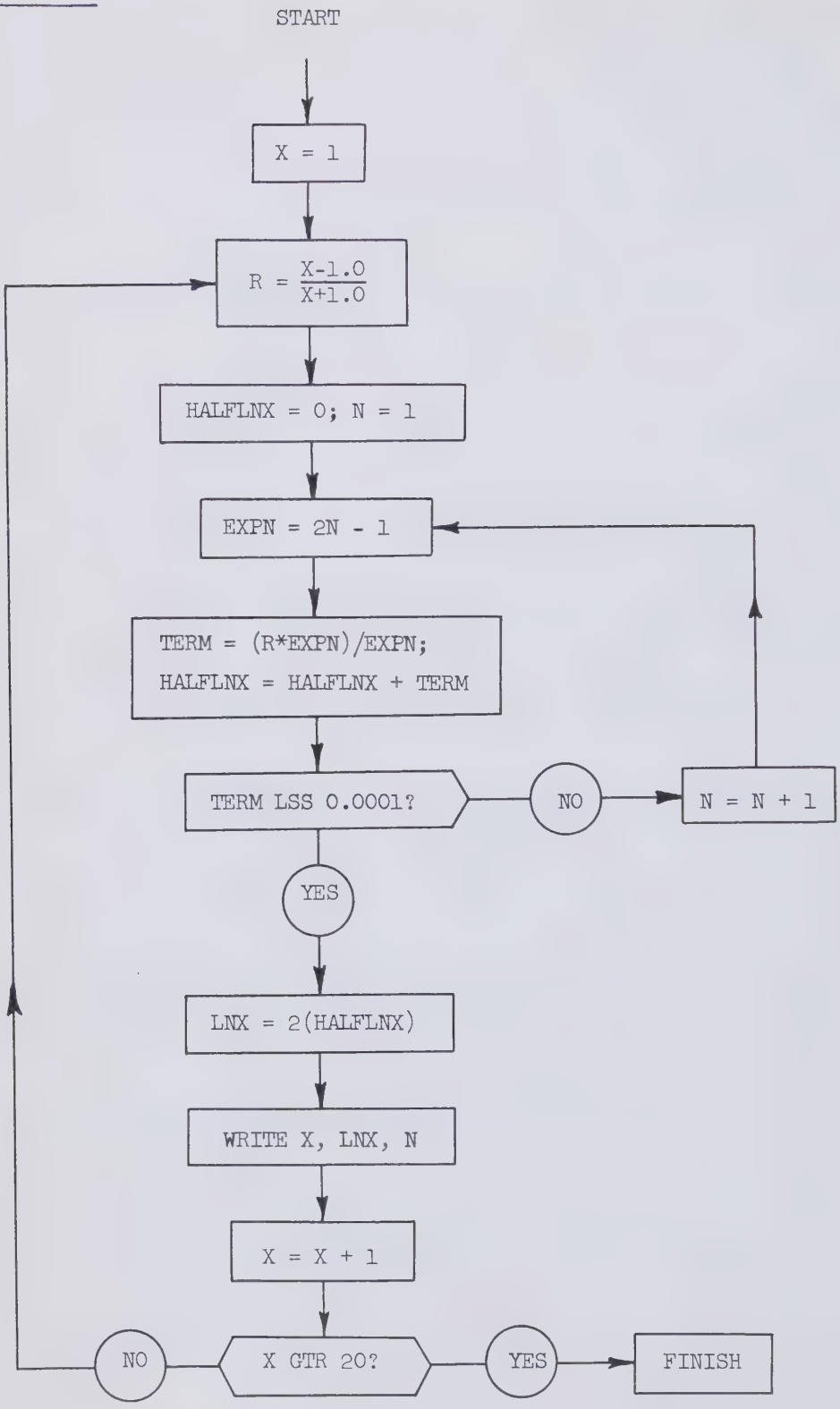
for values of X from 1 to 20 in steps of 1; that is, for X = (1,1,20). The latter notation for specifying an orderly sequence of values will be used in subsequent examples. The values within parentheses specify the initial value, the increment between successive values, and the upper limit, respectively, of the sequence of values that X is to assume.

The results obtained from the program are to be printed in three columns, the first containing values of the argument X, the second containing LN X with three digits to the right of the decimal point, and the third containing the number of terms N which were used in the series. The series is terminated when the value of the last term is less than 0.0001. The values of X and N are to be printed in integer form.

It should be noted first that the natural logarithm of a number can be obtained in a BALGOL program by means of the library procedure LOG. Hence the following example would not be used as part of an actual program; it is presented solely as a means of illustrating the language. The library procedures include SQRT, SIN, COS, TAN, SINH, etc., and are discussed in Art. 2.14 (p. 67). The general form of these procedures is the name of the procedure followed by the argument (a number, variable, or an expression) which is enclosed within parentheses; for example, LOG(X), SQRT(B*2 - 4A.C), SIN(Z/2), etc. These procedures can be included in arithmetic expressions in a program; for example, one can write Y = LOG(X), Z = LOG(X) + SIN(SQRT(B*2 - 4A.C)), etc.

As in previous examples, a flow chart for the program is presented first, then the BALGOL program, and then line-by-line comments on the program.

Flow Chart:


START
X = 1
R = (X-1.0)/(X+1.0)
HALFLNX = 0; N = 1
EXPN = 2N - 1
TERM = (R*EXPN)/EXPN;
HALFLNX = HALFLNX + TERM
TERM LSS 0.0001?
NO
N = N + 1
YES
LNX = 2(HALFLNX)
WRITE X, LNX, N
X = X + 1
NO
X GTR 20?
YES
FINISH

BALGOL Program	Card No.
Job Card	1
Lodx Card	2
INTEGER X, N;	3
X = 1;	4
L1.. R = (X - 1.0)/(X + 1.0);	5
HALFLNX = 0; N = 1;	6
L2.. EXPN = 2N - 1;	7
TERM = (R*EXPN)/EXPN;	8
HALFLNX = HALFLNX + TERM;	9
IF TERM LSS 0.0001;	10
BEGIN LNX = 2.HALFLNX;	11
WRITE (;; ØUT1, FØRM1);	12
ØUTPUT ØUT1 (X, LNX,N);	13
FØRMAT FØRM1 (B20,I3,B5,X6.3,B5,I4,WO);	14
X = X + 1;	15
IF X GTR 20; GØ TØ L3;	16
GØ TØ L1	17
END;	18
N = N + 1; GØ TØ L2;	19
L3.. FINISH;	20

Comments on Program:

Card 3. Type Declaration. This type declaration specifies that X and N are to be integer quantities. All other quantities are implicitly type floating.

Card 5. This assignment statement specifies that R is to be assigned the current value of the expression (X-1.0)/(X+1.0). The evaluation of this expression involves three arithmetic operations, namely, addition, subtraction, and division. Each such arithmetic operation involves two numeric operands (that is, values to be operated on). If both operands are type integer, then the operation will be performed using integer arithmetic. If both operands are type floating, or if one operand is type floating and one is type integer, the operation will be performed using floating-point arithmetic. In the latter case, the compiler provides for automatically transforming the integer operand to type floating prior to execution of the arithmetic operation.

In this example the number 1.0 is in decimal form and accordingly is treated as type floating. Therefore, the values of the arithmetic combinations X-1.0 and X+1.0 will be type floating, even though X is type integer. Thus, floating-point arithmetic will be performed when taking the ratio (X-1.0)/(X+1.0).

If the statement on Card 5 were written as

$$R = (X-1)/(X+1)$$

integer arithmetic would be performed, thereby producing the value zero for the variable R for all positive values of X.

Card 7. The variable EXPN is floating-point because it was so declared, even though the arithmetic expression 2N-1 consists of terms of type integer. The transformation in type is performed automatically before assignment is made.

Card 10. The IF statement beginning on this card consists of the IF clause followed by the compound statement on Cards 11 to 18. If the relationship TERM LSS 0.0001 is not satisfied, the compound statement is ignored and the next statement (N = N + 1) is executed. Thus, an additional term of the series is calculated. The series is terminated at the term that contributes less than 0.0001 to HALFLNX.

In this example, the number 0.0001 appears in decimal form in the arithmetic relation TERM LSS 0.0001 and hence is of the same type as the variable TERM, namely, type floating. However, it is not necessary that this always be the case. In general, the form of an arithmetic relation (see p. 13) is

$$E_1 \quad r \quad E_2$$

where E_1 and E_2 are arithmetic expressions and r is a relational operator (LSS, GTR, etc.). If the types of the values of E_1 and E_2 are not the same, then the value of the expression (E_1 or E_2) which is of type integer will be converted automatically to type floating before the relation is tested.

Card 11. Note that the multiplication operator could be omitted without ambiguity.

Cards 12 to 14. The WRITE statement, in combination with the OUTPUT and FORMAT declarations, causes the three variables X, LNX, and N to be printed in the format specified in FØRM1. The format is constructed of a string of editing phrases followed by an activation phrase W0. The editing phrases have the following meaning: B20 indicates 20 blank spaces; I3 indicates that an integer number (in this case the value of X) is to be printed in a 3-column field; B5 indicates 5 blank spaces; X6.3 indicates that one floating-point number (in this

case the value of LNX) is to be printed in a 6-column field with 3 digits to the right of the decimal point; B5 indicates 5 blank spaces; and I4 indicates that an integer (N) is to be printed in a 4-column field. The activation phrase WO specifies that the line described by the preceding phrases is to be printed with single spacing. (The zero is redundant and preferably is omitted to avoid confusion with the letter O.)

In this and preceding examples, a total of three of the six editing phrases have been illustrated. The three are the I phrase (for integer numbers), the X phrase (for floating-point numbers), and the B phrase for inserting blank spaces. Other phrases are F and S, for floating-point numbers, and the A phrase for alphanumeric information. For a description of the six phrases, see Art. 3.8 (p. 83). The use of the A phrase is illustrated in Art. 2.8 (p. 50).

Card 16. Note that this is an IF statement within an IF statement. Note also that it would not be correct to write this statement as

IF X GTR 20; FINISH;

since this would mean that the FINISH statement would not appear at the end of the program.

PROBLEMS

2.4-1. Write a program for preparing a table of squares and cubes of the numbers 1, 2, 3, ... , 10. Print the results in three columns (numbers, squares, and cubes) with the numbers appearing in integer form.

2.4-2. Write a program for preparing a table of factorials N! and reciprocal factorials 1/N! for N = (1,1,10). Print the results in three columns (N,N!,1/N!). The values of N and N! are to be printed in integer form and the values of 1/N! in floating-point form with eight digits to the right of the decimal point.

2.4-3. Assume that a set of 8-digit, integer numbers are available on consecutive data cards. Write a BALGOL program which will produce two new numbers corresponding to each of the original numbers such that the first number consists of digit positions 1, 3, 5, and 7 and the second number consists of digit positions 2, 4, 6 and 8. List the numbers in three columns (original number, digit-positions 1-3-5-7 number, and digit-positions 2-4-6-8). For example, if one of the numbers read from a card is 72942736 the corresponding output will be:

72942736 7923 2476

2.5 Comments and Headings

The programer may intersperse descriptive remarks within the BALGOL program by means of the COMMENT declaration if he so desires. Such remarks are solely for the programer's convenience and have no effect on the compiled machine-language program nor on its execution. As an example, one may wish to begin a program with a COMMENT declaration that describes briefly the purpose of the program. The declaration consists of the declarator COMMENT followed by any alphanumeric phrase. The term alphanumeric phrase refers to words or character groups that contain a combination of alphabetic, numeric, and special characters (including spaces). A complete list of these characters as defined in BALGOL is tabulated on p. 49 . In the COMMENT declaration the only exceptions are that there can not be a separator consisting of a pair of dots after the word COMMENT, since this confounds COMMENT .. with a statement label, nor can a semicolon be contained within the declaration. As in the case of any other declaration or statement, the separator ; follows the declaration.

As an example, the following line might have been inserted after Card 2 of the illustrative example on p. 31:

```
CØMMENT NATURAL LØGARITHMS ØF NUMBERS ;
```

This comment would appear in the listing of the BALGOL program in the same form in which it is written, thus identifying the program to the reader.

The BALGOL language also provides means for interspersing alphanumeric information in the output of the program. One way of accomplishing this is by using an alphanumeric-insertion phrase in a format contained in a FORMAT declaration. Such a phrase is enclosed within asterisks and is separated from adjacent editing or activation phrases by commas. The phrase itself may contain any alphanumeric characters (including spaces) other than an asterisk. The use of an alphanumeric-insertion phrase is particularly convenient in providing headings, titles, etc., on the printed output page.

The following illustration makes use of alphanumeric-insertion phrases:

```
WRITE (;; HEADING) ;

FØRMAT HEADING (B22,*X*,B6,*LN X*,B9,*N*,W,W4);
```

The execution of the WRITE statement causes the writing of the headings specified in the format. Note that in this case the WRITE statement does not refer to an output-data set since none is required, but refers only to a format. The format is as follows: 22 blank spaces,

an X, 6 blank spaces, the expression LN X (note that a space is considered to be a character), 9 blank spaces, and an N. The activation phrase W specifies that the preceding is to be printed on a single line after single spacing has occurred. The next phrase W4 specifies that the items to its left (there are none in this case) are to be printed after double spacing. Hence the effect of the two phrases together is to cause single spacing, printing of the desired heading, and then double spacing.

The two cards shown above could be inserted before Card 5 in the program shown on p. 31 . The result would be that the headings X, LN X, and N would appear above the columns of printed output information.

PROBLEMS

2.5-1. Write the statements and declarations which must be added to the illustrative example of Art. 2.1 (see p. 7) in order to print the headings A, B, C, D, and E over the printed columns of results.

2.5-2. Revise the illustrative example of Art. 2.2 (see p. 21) in order to provide for printing the headings N and RECIPROCAL FACTORIAL above the two columns of printed results.

2.5-3. Solve Problem 2.2-5 and print the results in the form

E = d.dddd N = ddd

where each d represents a digit.

2.6 Arrays and Subscripted Variables

As an illustration of a program containing an array and subscripted variables, let us consider the problem of generating a table that contains the integers 1 to 100, the squares of these integers, and the cubes of these integers; that is

1	2	3	4	...	100
1	4	9	16	...	10000
1	8	27	64	...	1000000

The table is an ordered array of integer values and could have any one of several possible arrangements; for example, the array may have 3 rows and 100 columns (as indicated above) or it may have 100 rows and 3 columns. In either case the subscripted variable X_{ij} furnishes a convenient method of referring to an element in the array. For instance, one can specify that i is an index over the rows of the array and j is an index over the columns of the array. Then X_{ij} becomes a subscripted variable referring to the value of the element in the ith row and jth column of the array (e.g., X_{23} refers to the value in the second row and third column of the array).

The printing device (the type 407 accounting machine) used with the Burroughs 220 does not provide for the printing of subscripts. To overcome this difficulty, the common algebraic form X_{ij} is represented as X(I,J) in BALGOL. The list of subscripts is enclosed in parentheses and the subscripts are separated by commas. This particular problem requires only two subscripts, but the language permits as many subscripts as may be required. The number of subscripts must be the same as the dimension of the corresponding array.

The flow chart shown in Fig. 4 and the BALGOL program which follow illustrate the computational process for generating the elements of the table illustrated above and storing them as a 3 by 100 array (3 rows and 100 columns) in memory. After the table is generated, it is printed as a 100 by 3 array (100 rows and 3 columns) with a heading over each column.

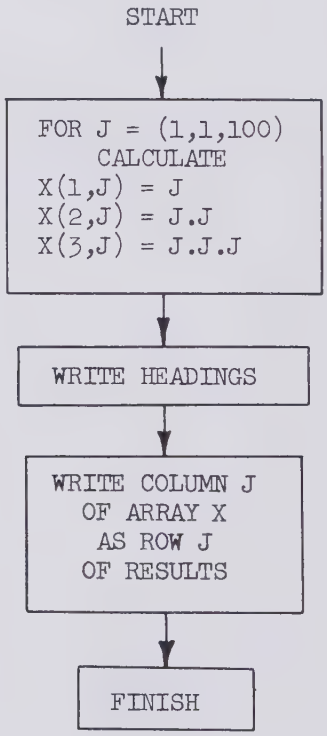


Fig. 4. Flow Chart

BALGOL Program	Card No.
Job Card	1
Lodx Card	2
CØMMENT PRØGRAM TØ COMPUTE SQUARES AND CUBES ØF INTEGERS FRØM 1 TØ 100;	3
INTEGER ØTHERWISE;	4
ARRAY X(3,100);	5
FØR J = (1,1,100);	6
BEGIN X(1,J) = J;	7
JSQ = X(2,J) = J.J;	8
X(3,J) = JSQ.J	9
END;	10
WRITE (;;HED);	11
FØRMAT HED (B23, *N*, B5, *SQUARE*, B6, *CUBE*, W, W4);	12
FØR J = (1,1,100);	13
WRITE (;; ØDS, FØRM);	14
ØUTPUT ØDS (FØR I = (1,1,3); X(I,J));	15
FØRMAT FØRM (B15, 3I10, W2);	16
FINISH;	17

Comments on Program:

Card 3. This COMMENT declaration identifies the program for the convenience of the programer.

Card 4. The type declaration INTEGER OTHERWISE declares that all variables in this program are to be considered as integers. (In general, this declaration specifies that all variables not otherwise declared explicitly as to type are automatically declared to be type integer.)

Card 5. ARRAY Declaration. Before any variable with subscripts may be used in a statement, it is necessary that the array of which it is an element be defined. This is accomplished by the ARRAY declaration (e.g., ARRAY X(3,100)) which in general consists of the declarator ARRAY and one or more array definitions that are separated from each other by commas. An array definition consists of an identifier (e.g., X) and a list of integers (e.g., 3,100) that defines the dimension and size of the array. This list is enclosed in parentheses and its elements are separated by commas. Also, the ARRAY declaration serves the purpose of indicating that the identifier X represents a

multi-valued variable. A subscripted variable (e.g., X(1,10)) refers to a single element in the array. Each integer in the list of integers in the array definition represents the upper limit for which the corresponding subscript is defined. Each upper limit must be two or greater, otherwise there is no need for that subscript. The lower limit of each subscript is automatically assigned the value one. The compiler automatically reserves a sufficient number of words in memory to store all elements of the array (in this case 300).

When explicitly declaring the type of an array identifier, it is sufficient to include the identifier in a type list (e.g., INTEGER X), although it is permissible to append parentheses to the identifier (e.g., INTEGER X()). The latter form is for the convenience of the programer since it emphasizes that X is an array identifier.

The type declaration for the identifier of the array must precede the ARRAY declaration. Thus, cards 4 and 5 in this example could not be interchanged.

In this example, values were assigned to the elements of the array by means of the assignment statements of Cards 7, 8, and 9. There are occasions, however, when the initial values of the elements in an array must be specified explicitly by the programer. This can be accomplished by reading the values from data cards and assigning them to the elements of the array by specifying an appropriate input-data set. A second method makes use of an alternative form of the array definition, and is discussed in Art. 3.6 (p. 78).

Cards 6 to 10. FOR Statement. Lines 6 to 10 in this example constitute a FOR statement. Line 6 contains a FOR clause and lines 7 to 10 contain a compound statement enclosed in the BEGIN ... END brackets. In general, a FOR statement comprises a FOR clause, a semicolon (acting as a separator), and a statement (either simple or compound). The function of the FOR clause is to control iteration (or repeated execution) of the statement that follows the clause.

In this particular example, the clause FOR J = (1,1,100) causes the compound statement contained in lines 7 to 10 to be executed repeatedly for values of J equal to 1, 2, 3, ... , 100. After this is done, the statement following the FOR statement is executed (Card 11). Thus the single FOR statement contained in lines 6 to 10 generates the required table of squares and cubes. The element (1,1,100) is called a step-until element and the details of how the element operates are discussed in Art. 3.5 (see p. 76).

In addition to step-until elements, a FOR clause may contain a sequence of numbers. For each such number, the statement contained in the FOR statement will be executed. Some examples of FOR clauses are as follows:

```
FOR K = 1, 3, 5, 7, 11, 13, 17
FOR K = (1,1,9), (10,5,50), 100, 200
FOR K = 2, 1.5, (1.0, -0.1, -1.0), -1.5, -2
```

These examples are self-explanatory and show that the numbers may be positive or negative, of any form (integer, decimal, or floating point), and that the step-until element may represent either an increase or decreasing sequence. Furthermore, any one of the numbers in the above examples may be replaced by an arithmetic expression, as discussed in Art. 3.5. Finally, it should be noted that if the type of the number is not the same as the type of the variable K, then the number will be converted to the type of K.

Card 7. Subscripted Variables. The variable X(1,J) is a subscripted variable, the subscripts specifying the variable to be the jth element of the first row in the array defined by the array declaration X(3,100). The statement X(1,J) = J is an assignment statement that assigns the current value of J to the variable X(1,J).

The general form of the subscripted variable is ID(E_1,E_2, ... ,E_n), where ID represents an arbitrary identifier and E_1, E_2, ... , E_n represent arithmetic expressions. The identifier must have been declared previously (either explicitly or implicitly) as to type, and the array of which the subscripted variable is an element must have been explicitly declared previously as to size. As mentioned above, this is accomplished by the ARRAY declaration, e.g., ARRAY ID(U_1, U_2, ... , U_n), in which U_1, U_2, ... , U_n must be positive integers greater than one that specify the upper limits of the corresponding subscripts. A subscript is redundant when its upper limit is one. The lower limit of each subscript is automatically taken as one. Thus, the subscript expressions E_1, E_2, ... , E_n must satisfy the following restrictions: (1) the value of a subscript must not be less than 1; and (2) the value of a subscript must not exceed its upper limit as specified in the ARRAY declaration. If either of these restrictions is not satisfied, the consequences are undefined, but may lead to erroneous results.

A subscript expression E_i may have decimal or floating values, such values being automatically truncated to an integer. For example, the portion of the number 2.356 to the right of the decimal point would be removed in truncation leaving the integer 2 as the value of the subscript expression.

Card 8. Generalized Assignment Statement. As indicated in this example, two or more variables may be assigned the same value in a single assignment statement. Of course, if all the variables are not of the same type, one must use care in the formulation of a generalized assignment statement. Consider first the simple assignment $V_1 = E_1$. If the declared type of V_1 differs from the type of E_1, the value obtained from the evaluation of E_1 will be converted automatically to the type specified for V_1 and the converted value will be assigned to V_1. In particular, a number of type floating will be truncated to an integer as described above.

Consider next a generalized assignment statement $V_2 = V_1 = E_1$, where E_1 is an expression which yields a number of type floating, V_1 is type integer, and V_2 is type floating. The result of this generalized assignment would be that the value of E_1 would be truncated to an integer for assignment to V_1, and then the integer value would be converted to a floating value and assigned to V_2. However, the decimal-fraction portion of the value of E_1 would have been truncated in the first assignment. If, instead, the statement were written $V_1 = V_2 = E_1$, then V_2 would be assigned the value of E_1 and V_1 would be assigned the truncation of the value of V_2.

In this example the generalized assignment statement and the variable JSQ are introduced into the program solely in order to save computer time during execution. It is more efficient to assign the value of X(2,J) to JSQ and then use JSQ in the next calculation (Card 9) than it is to calculate the relative location in the array of the value of X(2,J) for use in the latter calculation, as would be required if the program were written as follows:

```
X(2,J) = J.J;

X(3,J) = J.X(2,J);
```

Cards 11 and 12. This WRITE statement in combination with the FORMAT declaration causes the printing of a line of headings for the table of results. The appearance of the line of headings is as follows: 23 blank spaces, the letter N, 5 blank spaces, the word SQUARE, 6 blank spaces, and the word CUBE. The spacing has been so arranged that the headings will appear above the appropriate columns of results.

Cards 13 and 14. This is a FOR statement which causes 100 executions of the WRITE statement on Card 14. The WRITE statement causes printing of the output-data set ODS (see Card 15) in the format FORM (see Card 16).

Card 15. OUTPUT Declaration with Subscripted Variables. The FOR clause in the output-data set indicates that the set consists of three variables X(1,J), X(2,J), X(3,J). In other words, the BALGOL language permits the use of the FOR clause to facilitate specification of a sequence of subscripted variables in an output-data set. Note that the FOR clause is followed by a subscripted variable, rather than by a statement, when used in this manner.

The FOR clause may also be used with an INPUT declaration in an analogous manner.

An alternative scheme for use in this illustrative program would be to omit Card 13 and replace Card 15 by the following:

```
ØUTPUT ØDS(FØR J = (1,1,100); FØR I = (1,1,3); X(I,J));
```

If this is done, no change in the FORMAT declaration is needed, since a format enclosed in parentheses (as in Card 16) will be repeated indefinitely for the corresponding output-data set.

PROBLEMS

2.6-1. Solve Prob. 2.2-4 by making use of a FOR statement.

2.6-2. Solve Prob. 2.4-1 by making use of a FOR statement and subscripted variables.

2.6-3. Solve Prob. 2.4-2 by making use of a FOR statement and subscripted variables.

2.6-4. Prepare three data cards, each containing six decimal numbers. The numbers on the first card are to be identified as $p_1, p_2, \cdots, p_6$; on the second card, $q_1, q_2, \cdots, q_6$; and on the third card, $r_1, r_2, \cdots, r_6$. Write a BALGOL program which will print a 5 by 6 array of numbers having the values

p_1	p_2	$\cdots$	p_6
q_1	q_2	$\cdots$	q_6
r_1	r_2	$\cdots$	r_6
p_1q_1	p_2q_2	$\cdots$	p_6q_6
q_1r_1	q_2r_2	$\cdots$	q_6r_6

2.6-5. Write a program for multiplying a 4 x 5 matrix [A] by a 5 x 1 column matrix [B], the result being a 4 x 1 column matrix [C]. Assume that the elements of matrices [A] and [B] are to be read from cards in the following manner: data card 1 contains in sequence the four elements in the first column of [A], data card 2 contains the elements in the second column of [A], etc., and data card 6 contains the elements of [B]. Arrange the results on the printed page in the following manner (with headings):

MATRIX A				MATRIX B	MATRIX C
A_{11}	A_{12}	...	A_{15}	B_1	C_1
A_{21}	A_{22}	...	A_{25}	B_2	C_2
A_{31}	A_{32}	...	A_{35}	B_3	C_3
A_{41}	A_{42}	...	A_{45}	B_4	C_4
				B_5	

2.6-6. Write a program for multiplying two 4 x 4 square matrices [A] and [B], assuming that the elements of each row of [A] and [B] are to be read from punched cards.

2.7 Iterations Controlled by the UNTIL Statement

Iterative processes occur very frequently in numerical calculations and hence control of iterations is an important part of any computer language. In the examples of Articles 2.1, 2.2, and 2.4, it was shown that an IF statement could be used to control an iterative process by specifying that a statement was to be performed conditionally. Then in the preceding article the FOR statement was introduced as a means of specifying that a statement was to be performed for a specified set of values of a variable. Still another means of controlling an iteration is the UNTIL statement. The UNTIL statement causes a statement to be executed repeatedly until a specified condition is satisfied.

As an illustration of the use of the UNTIL statement, consider the following program for calculating the square root of a number by an iterative process. It should be noted first that the square root of a number can be obtained in a BALGOL program by means of the library procedure SQRT (see Art. 2.14). Hence the example which follows would not be used as part of an actual program. It is selected solely because it provides a very simple example of an iterative process.

Assume that it is desired to calculate values of the variable Y equal to the square root of another variable X, that is

$$Y = \sqrt{X} \qquad \text{and} \qquad Y^2 = X$$

From these relations it is apparent that

$$2Y^2 = Y^2 + X \qquad \text{and} \qquad Y = \frac{1}{2}\left(Y + \frac{X}{Y}\right)$$

This suggests a recursive formula

$$\text{YNEW} = \frac{1}{2}\left(\text{YOLD} + \frac{X}{\text{YOLD}}\right) \qquad (1)$$

The recursive formula provides a means of calculating Y for any positive value of X by iteration. Beginning with the initial approximation for Y, called YOLD, the right-hand side of Equation (1) is evaluated. This produces a new approximation for Y, called YNEW. Then this new approximation is treated as YOLD and substituted again in the right-hand side of the equation, thus giving a still more accurate approximation for YNEW. This process is continued until the difference between YNEW and YOLD has reached a sufficiently small value, say less than a small number epsilon (EPS).

The following statements are assumed to be part of a complete BALGOL program. It is assumed that all variables are type FLOATING and that elsewhere in the program a value has been assigned to X. It is then desired to obtain the square root of X, if X is positive or zero. If X is negative, it is desired to branch to another part of the program.

```
                                                        Card No.
EITHER IF X LSS 0; GØ TØ L2;                              1
ØR IF X EQL 0; BEGIN YØLD = 0; GØ TØ L1 END;              2
ØR IF X GEQ 1; YØLD = X/2;                                3
OTHERWISE; YØLD = 2X;                                     4
EPS = 0.00001;                                            5
DIFF = 1;                                                 6
UNTIL DIFF LEQ EPS;                                       7
      BEGIN YNEW = 0.5 (YØLD + X/YØLD);                   8
            DIFF = ABS (YØLD - YNEW);                     9
            YØLD = YNEW                                  10
      END;                                               11
L1.. Y = YØLD;                                           12
```

Comments on Program:

Cards 1 to 4. EITHER IF Statement. The statement on these cards constitutes a single EITHER IF statement, which may be considered as a generalization of the IF statement. The reserved word EITHER indicates the start of a special sequence of IF statements, while the reserved word OR indicates a continuation of the sequence. Beginning with the first IF clause, each clause is tested in sequence to determine whether or not its condition is satisfied. When a clause is encountered whose condition is satisfied, the statement immediately following it is executed. Then the entire EITHER IF statement is considered to be satisfied, and the next statement to be executed is the one following the statement (Card 5 in this example), unless a GO TO statement has caused a branch to some other part of the program. If none of the IF clauses is satisfied, the statement following OTHERWISE is executed.

Thus, in this example, if the relation X LSS 0 is satisfied, the statement GO TO L2 is executed. If it is not satisfied, but the relation X EQL 0 is satisfied, then the compound statement on Card 2 is executed. If neither of the first two relations is satisfied, but the relation X GEQ 1 is satisfied, then the assignment statement YOLD = X/2 is executed, after which the statement on Card 5 is executed. If all three IF clauses are not satisfied, the statement YOLD = 2X is executed, after which the statement on Card 5 is executed.

Another form of the EITHER IF statement is discussed in Art. 3.4. In the reference manual, Burroughs Algebraic Compiler, the term alternative statement is used as a name for the EITHER IF statement.

Card 5. The value of the quantity EPS must be specified.

Card 6. An initial value for DIFF must be specified (greater than EPS) or else the UNTIL statement may not be executed at all. For example, if DIFF is not assigned a value, it is possible that its current value is zero. Then the condition DIFF LEQ EPS (Card 7) is satisfied initially and the UNTIL statement (Cards 7 to 11) is not executed.

Cards 7 to 11. UNTIL Statement. These cards contain an UNTIL statement, consisting of an UNTIL clause (Card 7) followed by a compound statement (Cards 8 to 11). The compound statement is executed repeatedly until the condition specified in the UNTIL clause is satisfied. Then the next statement following the UNTIL statement is executed.

To be more specific, the general form of the UNTIL statement is

UNTIL C; S

The letter C indicates a condition (e.g., a relation) and the letter S indicates a statement (either simple or compound). The UNTIL statement operates in the following manner. First, the condition C in the UNTIL clause is tested and one of two results obtains. Either the condition is satisfied (e.g., DIFF $\leq$ EPS) or else the condition is not satisfied (e.g., DIFF > EPS). Second, if the condition is satisfied, then the statement S following the UNTIL clause and contained in the UNTIL statement is not executed. Instead, in this example the next statement is executed (Card 12). Third, if the condition is not satisfied, then the statement S is executed. Then the condition is tested again and the cycle just described is repeated until the condition is satisfied. Another possibility is that the statement S contains a GO TO statement, in which case the cycle is broken if that statement is executed. This is illustrated by the flow chart in Fig. 5.

Another way of writing the UNTIL statement in the preceding example is to omit Card 5 and replace Card 7 by the following:

UNTIL DIFF LEQ 0.00001;

In addition, the condition C in an UNTIL clause may consist of a Boolean expression as described previously for the IF clause (see p. 22). For example, the condition may consist of two arithmetic relations, each enclosed in parentheses, and separated by a Boolean operator, such as OR or AND. Thus, one can write

(N GTR 100) OR (X LSS EPS)

as the condition in an UNTIL clause. Boolean expressions are described in Art. 3.2 (p. 71).

Card 9. Intrinsic Function ABS. This is one of the six intrinsic functions that are part of the BALGOL language. The general form of this function is ABS(E), where E is a number, variable, or an arithmetic expression. The value of ABS(E) is the absolute value (magnitude) of E.

The other intrinsic functions are MOD, MAX, MIN, SIGN, and PCS. For a description of these functions, see Art. 2.11 (p. 57).

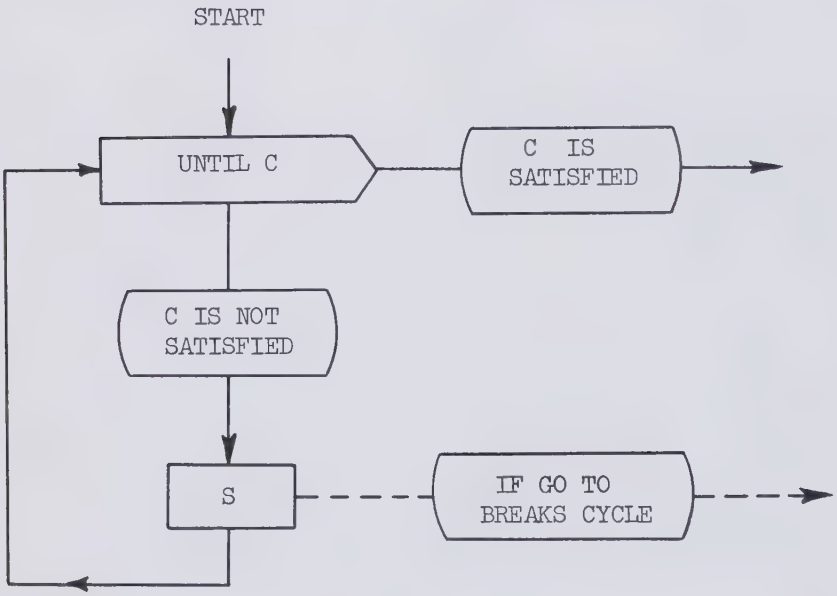


Fig. 5. Flow Chart for UNTIL Statement

PROBLEMS

2.7-1. Solve Problem 2.2-5, by making use of an UNTIL statement.

2.7-2. Solve Problem 2.2-6 by making use of an UNTIL statement.

2.7-3. Write a program to calculate the value of π from the series

$$\frac{\pi^2}{8} = 1 + \frac{1}{3^2} + \frac{1}{5^2} + \frac{1}{7^2} + \cdots$$

2.7-4. Modify the square root program used as an illustrative example in this article to calculate imaginary roots for negative values of X.

2.7-5. An iterative method for determining the roots of an algebraic equation is illustrated by the following problem. Assume that we have the equation

$$X^3 - 5X^2 + 6X - 1 = 0$$

Rewrite the equation in the form

$$X = \frac{1 + 5X^2 - X^3}{6}$$

which gives the recursive formula

$$X_{i+1} = \frac{1 + 5X_i^2 - X_i^3}{6}$$

Then, from the latter equation, calculate an approximate value for X_{i+1} by substituting into the right-hand side a trial value of X_i. Then using the value of X_{i+1} as the new value of X_i, repeat the process.

Write a BALGOL program for calculating the smallest positive root of the above equation to an accuracy of 6 significant figures. (Suggestion: Take X = 0 on right-hand side of equation as first trial value.) Since this method for calculating roots may converge slowly, include in the program a provision for limiting the number of iterations to 10. List the consecutive approximations to the root in the output of the program. (Answer: X = 0.198062)

2.8 Alphanumeric Input and Output

The BALGOL language provides for the processing of alphabetic and special characters as well as numeric characters. The term alphanumeric is used to refer to words or character groups that may contain a combination of such characters. It was shown previously that alphanumeric information could be inserted in the program listing by means of the COMMENT declaration, and that such information could be interspersed within the output data by means of the alphanumeric-insertion phrase in a format. The latter provision is particularly useful for titles and headings.

In addition to the provisions listed above, it is possible to read alphanumeric information from data cards and also to have such information printed as output. In order to understand how this is accomplished, it is first necessary to observe how alphanumeric characters are coded for storage in the memory of the computer. Since information is stored in memory in the form of 10-digit words, each alphanumeric character is assigned an integer equivalent before it can be stored in memory. Since there are 47 characters to be coded, the integers 00 through 99 more than suffice to provide a unique code consisting of a two-digit integer to represent each alphanumeric character. The convention shown in Table 1 has been adopted for the Burroughs 220.

Since the numeric equivalent of each alphanumeric character requires two digit positions, and since each word in memory contains 10 positions, it follows that each computer word can contain the equivvalent of no more than five alphanumeric characters.

It is necessary to adopt a convention that distinguishes alphanumeric information on a data card from numeric information. The BALGOL language requires that alphanumeric information on a data card be enclosed within semicolons (hence a semicolon cannot be contained within the alphanumeric information itself). For example, the following might appear on a data card:

```
;PROBLEM NO. 37; 72.0 1914.2
```

The 14 characters within the semicolons would be treated as alphanumeric information; the two numbers following would be treated as numerical data. When these data are read, the alphanumeric characters between semicolons must be divided into words not exceeding 5 characters each. A space is counted as a character. The first word consists of the first five characters, reading left to right. The second word consists of the next five characters, etc. If the last word contains fewer than five characters, it is automatically filled out with spaces, for which the numeric code is 00.

TABLE 1. INTEGER EQUIVALENT OF ALPHANUMERIC CHARACTERS

Alphanumeric Character in BALGOL	Integer Equivalent
A to I	41 to 49
J to R	51 to 59
S to Z	62 to 69
0 to 9	80 to 89
+	10
-	20 or 34
=	33
(	24
)	04
.	03
,	23
; or $	13
/	21
*	14
Space	00

As in the case of numeric values, a variable must be specified to which the value of each word will be assigned when such data are read. It will be recalled that a list of such variables is specified in the definition of an input-data set contained in an INPUT declaration.

In the example given above the alphanumeric group PROBLEM NO. 37 contains 14 characters (assuming one space after PROBLEM and one space after NO.). Therefore, three identifiers are required in the input-data set. For example, the INPUT declaration corresponding to the data shown above might be:

```
INPUT IDS(ALPH1, ALPH2, ALPH3, X, Y)
```

where ALPH1 is an identifier corresponding to the first five characters, ALPH2 corresponds to the next five, and ALPH3 corresponds to the last four characters, with a space added automatically to complete the group. The variable X would be assigned the value 72.0 and Y would be assigned 1914.2. The identifier IDS refers to the input-data set and would appear in the associated READ statement; for example, READ (;;IDS).

The reading of the alphanumeric information PROBLEM NO. 37 from a data card would result in the following numbers being assigned to ALPH1, ALPH2, and ALPH3, respectively:

5759564253 4554005556 0300838700

If such values are to be operated on for any purpose other than reading or writing, their identifiers should be declared as type integer.

An alternative way of specifying the identifiers in the INPUT declaration illustrated above is to replace the simple variables ALPH1, ALPH2, ALPH3, with subscripted variables, as follows:

```
INPUT IDS(FOR K = (1,1,3); ALPH(K), X,Y)
```

This form is useful if the alphanumeric string to be read into the computer is lengthy. In the example given, the INPUT declaration must be preceded by the ARRAY declaration ARRAY ALPH(3).

Alphanumeric output of information requires a WRITE statement, an OUTPUT declaration, and a FORMAT declaration. In the WRITE statement, alphanumeric information is distinguished from numeric information by an alphanumeric editing phrase (the A phrase) in a format definition. The WRITE statement causes the output-data set identified by the OUTPUT declaration to be printed in the format specified in the FORMAT declaration. Consequently, the identifiers associated with the alphanumeric information must appear in the OUTPUT declaration, and the corresponding editing phrase in the format definition is distinguished by the letter A.

To continue the example, the following portion of a program could be used to cause printing of the information read from the foregoing data card (except for the two semicolons):

```
WRITE (;; ØDS, FØRM);
ØUTPUT ØDS (ALPH1, ALPH2, ALPH3, X,Y);
FØRMAT FØRM (2A5, A4, B6, 2X10.1,W);
```

The editing phrase 2A5 specifies that ALPH1 and ALPH2 are to be printed as alphanumeric phrases, each of five characters. The phrase A4 specifies that four characters of ALPH3 are to be printed.

These first two editing phrases could be replaced with the single editing phrase A14 and the same results would be achieved. Another alternative is to replace the first three phrases with the phrases A15, B5.

The remaining phrases in the format are associated with editing the values of X and Y, which are 72.0 and 1914.2, respectively, and with printing the edited line.

The use of a FOR clause in the OUTPUT declaration is also permitted, analogous to its use in an INPUT declaration.

PROBLEMS

2.8-1. Write a BALGOL program which will read your name and street address from á single data card and will then print the same information on two output lines (name on first line, address on second line).

2.8-2. Write a BALGOL program which will read the alphabet (A,B, ... ,Z) from one data card and then will print two lines of results, the first containing the alphabet in usual order (with a space between letters) and the second containing the alphabet in reverse order.

2.9 Subroutines

There are many computational processes that have a wide range of applicability. For example, the need for solving a set of simultaneous linear equations, for multiplying matrices, for inverting a matrix, etc. recurs in a wide variety of problems. It is desirable that an algorithmic language provide for general purpose descriptions of such frequently used computational processes in order that needless duplication of programing effort may be obviated. The subroutine, the function, and the procedure can be used in a variety of ways to accomplish this purpose. These three types of subprograms will be discussed in this and subsequent articles.

The subroutine is useful for intraprogram reduction in programing effort. It is basically a compound statement with a name rather than a label. It can be used effectively to eliminate repetition of the same compound statement at various points in a single program.

The subroutine is defined by a declaration that consists of the declarator SUBROUTINE, an identifier which serves as a name for the subroutine, and a compound statement. The identifier and the statement are separated by a semicolon. Thus the general form for a subroutine is

SUBROUTINE ID; BEGIN S_1, S_2, ... , S_n END

At least one of the statements in the compound statement must be a RETURN statement, which will be discussed later.

Execution of a subroutine is initiated by execution of an ENTER statement which has the form

ENTER ID

in which ID represents the identifier of the subroutine.

The following example is presented as an illustration of a subroutine in a BALGOL program. The purpose of the subroutine is to calculate the square root of a variable X to which a value has been assigned elsewhere in the program. The method of calculating the square root of X is described earlier in Art. 2.7 (see p. 43) and the sequence of statements given on p. 44 is incorporated in the subroutine which follows. It is assumed that all variables are type FLOATING. If X is positive or zero, the square root is calculated, but if X is negative, a GO TO statement is executed which causes a branch to another part of the main program.

The declaration of the subroutine is as follows:

```
                                                          Card No.
SUBRØUTINE SQUARERØØT;                                        1
   BEGIN                                                      2
      EITHER IF X LSS 0; GØ TØ L2;                            3
      ØR IF X EQL 0; BEGIN YØLD = 0; GØ TØ L1 END;            4
      ØR IF X GEQ 1; YØLD = X/2;                              5
      ØTHERWISE; YØLD = 2X;                                   6
      EPS = 0.00001; DIFF = 1;                                7
      UNTIL DIFF LEQ EPS;                                     8
           BEGIN YNEW = 0.5 (YØLD + X/YØLD);                  9
               DIFF = ABS (YØLD - YNEW);                     10
               YØLD = YNEW                                   11
           END;                                              12
      L1.. Y = YØLD;                                         13
      RETURN;                                                14
   END SQUARERØØT;                                           15
```

Comments on Program:

Card 1. The declarator SUBROUTINE indicates the start of a subroutine declaration. It is followed by the arbitrarily-selected name of the subroutine, in this case SQUAREROOT.

Cards 2 to 15. The body of the subroutine is a compound statement enclosed with BEGIN ... END brackets. The calculations given here for the square root of X are described in Art. 2.7.

Card 14. The RETURN statement consists of the single word RETURN. After the RETURN statement is executed, the next statement in the main program following the ENTER statement that called the subroutine will be executed.

Card 15. As illustrated here, the subroutine identifier may follow the final word END if desired.

To illustrate how the preceding subroutine is called, let us assume that the following sequence of statements is part of a program containing the subroutine declaration SQUAREROOT:

```
                                                    Card No.
...                                                    1
...                                                    2
X = (A - 2)/LØG(2A);                                   3
ENTER SQUARERØØT;                                      4
IF Y GTR B*2; ...                                      5
...                                                    6
...                                                    7
X = LØG(B) + 1/B*2;                                    8
ENTER SQUARERØØT;                                      9
...                                                   10
...                                                   11
L2.. WRITE (;; MESSAGE);                              12
...                                                   13
```

In the above partial program it is assumed that X is assigned a value (Card 3). Then the ENTER statement (Card 4) calls the subroutine SQUAREROOT, which must be contained in the program but need not appear necessarily prior to the ENTER statement. After the RETURN statement in the subroutine is executed, the next statement to be executed is the statement (Card 5 in the main program) which follows the ENTER statement. The RETURN statement will not be executed if the value of X is negative, since the GO TO L2 statement in the subroutine causes a branch to the statement labeled L2. This might be a WRITE statement which causes printing of a message (see Card 12).

Note that an identifier which is used in the subroutine must be defined in the same manner as an identifier in the main program and it will have the same meaning in both places. Thus, the subroutine is in every sense a component part of the program. Just as compound statements may be contained within compound statements, subroutines may be defined within other subroutines.

2.10 Functions

The FUNCTION declaration is basically an assignment statement and can be used to eliminate repetition of the same assignment statement at various places in a single program. It may be used for either intra-program or interprogram reduction in programing effort. It may be adapted for use in several programs since the identifiers used as parameters within the FUNCTION declaration are semi-independent of those used elsewhere in the program.

A function is defined by a declaration that consists of the declarator FUNCTION followed by an assignment statement. The general form is

$$\text{FUNCTION ID}(P_1, P_2, \ldots, P_n) = E$$

The identifier ID chosen as a name for the function is followed by a list of formal value-parameters enclosed in parentheses and separated from each other by commas. The formal value-parameters are identifiers of simple variables which serve as the parameters of the function. The expression E on the right-hand side of the assignment symbol is usually formulated in terms of constants and the variables in the formal value-parameter list. If desired, any other variables appearing in the program containing the function declaration may appear in the expression; however, this makes the function dependent upon the rest of the program.

As mentioned above, the variables appearing in the formal value-parameter list are semi-independent of identifiers used elsewhere in the program. This is because they must be declared as to type in the main program. However, an identifier that is used in the formal value-parameter list may be used for a different purpose in the main program (i.e., outside of the function declaration). The only restriction is that the two different uses must require only one type declaration for that identifier.

Evaluation of a function occurs whenever its function designator appears in an expression in the program. The function designator consists of the function identifier followed by a list of actual value-parameters that are enclosed in parentheses and separated from each other by commas:

$$\text{ID}\ (E_1, E_2, \ldots, E_n)$$

The actual value-parameters are expressions defined in terms of constants and/or variables that appear in the program. When a function designator is encountered in program execution, each expression in the actual value-parameter list is evaluated and its current value is assigned to the corresponding variable in the formal value-parameter list prior to evaluation of the expression defined in the function declaration.

It is important to note that a function must be declared before it is referred to by a function designator. Also, a function may be declared inside of a subroutine.

To illustrate the FUNCTION declaration, let it be assumed that in a certain program the inverse hyperbolic sine of a number must be evaluated several times. Hence the following FUNCTION declaration is made a part of the program:

```
FUNCTION ARCSINH(X) = LOG(X + SQRT(X*2+1))
```

Now let it be assumed that at a later point in the program the inverse hyperbolic sine of a variable BUG is to be calculated and its value assigned to another variable BOG. The following statement could be used:

```
BOG = ARCSINH(BUG)
```

The current value of the actual value-parameter BUG is assigned to the formal value-parameter X in the FUNCTION declaration and then the function ARCSINH(X) is evaluated. The resulting value is then assigned to the variable BOG.

As another example, let it be assumed that the assignment statement

```
X = ARCSINH (ARCSINH(Y*2+1))
```

appears elsewhere in the program. In this case two successive evaluations of the declared function would occur. First, the expression Y*2+1 is evaluated and its value assigned to the formal value-parameter X in the declared function ARCSINH. The resulting value of ARCSINH(X) is then reassigned to the formal value-parameter X and the function evaluated a second time. This value is then assigned to the variable X in the main program. In this case the variable X appears in the FUNCTION declaration and also in the assignment statement, with different meanings in each case. A single type declaration (either explicit or implicit) in the program must satisfy both meanings.

Intrinsic functions are discussed in the next article and functions defined by procedures are discussed in Art.2.13 (p. 64).

2.11 Intrinsic Functions

There are six intrinsic functions that are part of the BALGOL language and can be designated without the necessity of a FUNCTION declaration. One of these functions (ABS) was encountered in a previous article (see p. 46). The others are MOD, MAX, MIN, SIGN, and PCS. The evaluation of one of these functions occurs whenever the corresponding function designator appears in an expression in the program.

The designator for the function ABS has the form ABS(E), where E is an arithmetic expression (of which numbers and variables are special cases). The value of the function ABS(E) is the absolute value, or magnitude, of E and its type (integer or floating) is the same as that of E itself. The function designator may appear in any arithmetic expression in the program; for example, one can write the assignment statement

```
Y = ABS(A*2 - B) + ABS(SQRT(C - ABS(E)))
```

or the output-data set

```
X, Y, ABS(Z)
```

The function designator may appear also in a Boolean expression, as illustrated by the following UNTIL clause:

```
UNTIL (ABS(X) LSS 0.001) OR (N GTR 50)
```

The general form of the MOD function is

$$\text{MOD}(E_1, E_2)$$

where E_1 and E_2 may be numbers, variables, or arithmetic expressions, but their values must be type integer. The type of the MOD function is also integer, and the value of the function MOD is the integer obtained as the remainder when the value of E_1 is divided by the value of E_2. For example, MOD(98,10) equals 8 since 98/10 equals 9 plus 8/10; MOD(72,5) equals 2 since 72/5 equals 14 plus 2/5; MOD(24,72) equals 24 since 24/72 equals 24/72; MOD(-72,5) equals -2 since -72/5 equals -14 plus -2/5; MOD(72,-5) equals 2 since 72/-5 equals -14 plus 2/(-5); etc.

In arithmetic two integers are said to be congruent modulo m if they differ by an integral multiple of m. For example, 8 and 98 differ by an integral multiple of 10; therefore, 8 is congruent to 98(mod 10), or more concisely,

$$8 \equiv 98 \pmod{10}$$

This implies also that

$$98 \equiv 8 \pmod{10}$$

From this definition of congruence it is seen that, in general, the intrinsic function MOD has the following meaning:

$$MOD(E_1, E_2) \equiv E_1 (\text{mod } E_2)$$

The MAX and MIN functions are of the form $MAX(E_1, E_2, \ldots, E_n)$ and $MIN(E_1, E_2, \ldots, E_n)$, respectively, in which there must be at least two expressions in the list ($n \geq 2$). The value of MAX is that of the algebraically largest expression in the list and the value of MIN is that of the algebrically smallest expression. The type of the function is integer if all of the expressions are integer; otherwise, the type is floating.

The function SIGN has the form SIGN(E), where E is an expression, and is equal to +1 if E is positive (when evaluated), -1 if E is negative, and 0 if E is zero. The type of the function is the same as that of E.

All of the above functions may be used in any arithmetic or Boolean expression in a program, as illustrated for the function ABS.

The function PCS has the form PCS(E) and is used to determine whether a particular Program Control Switch is on or off. The use of this function is described in the reference manual Burroughs Algebraic Compiler.

2.12 Procedures

The PROCEDURE is a subprogram which is particularly useful for eliminating duplication of programing effort. Its importance stems from the fact that the identifier notation used within a procedure is completely independent of the notation used outside the procedure. This feature makes it quite simple for a programer to incorporate in his program a procedure written by another programer. Except for the identifier chosen as the name of the procedure itself, the programer may duplicate in his program any identifiers used in a procedure that is to be incorporated. The method for achieving communication between a program and a procedure contained therein will be explained in the discussion of the example that follows.

The example defines a procedure for performing the computations required in matrix-by-vector multiplication. The column vector V, having the form

$$\begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_{RVMAX} \end{bmatrix}$$

is to be premultiplied by the matrix M which has RMAX rows and CMAX columns:

$$\begin{bmatrix} m_{1,1} & m_{1,2} & \cdots & m_{1,CMAX} \\ m_{2,1} & m_{2,2} & \cdots & m_{2,CMAX} \\ \cdots & \cdots & \cdots & \cdots \\ m_{RMAX,1} & m_{RMAX,2} & \cdots & m_{RMAX,CMAX} \end{bmatrix}$$

It is, of course, necessary that the number of columns of M be the same as the number of rows of V in order that M and V are compatible for matrix-by-vector multiplication, that is, CMAX = RVMAX. The result of the multiplication M·V is a product matrix P which has the form

$$\begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_{RVMAX} \end{bmatrix}$$

in which

$$p_1 = m_{1,1}v_1 + m_{1,2}v_2 + \cdots + m_{1,CMAX}\,v_{RVMAX}$$

$$p_2 = m_{2,1}v_1 + m_{2,2}v_2 + \cdots + m_{2,CMAX}\,v_{RVMAX}$$

$$\vdots$$

In order that the multiplication procedure be generally applicable, it is necessary that the size of the matrix M be variable, hence the procedure is defined in terms of the number of rows RMAX and the number of columns CMAX. In the event that CMAX is not equal to RVMAX, the procedure provides an immediate exit from the procedure to a labeled statement in the main program.

The declaration for a matrix-by-vector multiplication procedure named MATVECMULT is as follows:

```
                                                                  Card No.
PRØCEDURE MATVECMULT(RMAX,CMAX,RVMAX;M(,),V(),P(); ERR);             1
    BEGIN                                                             2
      INTEGER RMAX,CMAX,RVMAX,R,C;                                    3
      IF CMAX NEQ RVMAX; GØ TØ ERR;                                   4
      FØR R = (1,1,RMAX);                                             5
          BEGIN                                                       6
            SUM = 0;                                                  7
            FØR C = (1,1,CMAX);                                       8
               SUM = SUM + M(R,C)V(C);                                9
            P(R) = SUM                                               10
          END;                                                       11
    RETURN END;                                                      12
```

The PROCEDURE declaration is a self-contained description of the computational process and consists of the declarator PROCEDURE, a procedure heading, a semicolon, and the procedure body. In this example the heading appears on Card 1 following the declarator, and the body appears on Cards 2 to 12. The heading includes the procedure identifier (MATVECMULT) and the formal-parameter list which is contained in parentheses. The formal-parameter list consists of those identifiers and labels used in the description of the computational process that are to serve as channels of communication between the procedure and a program which contains it. The procedure body is a compound statement. It must contain declarations that are necessary

to define the properties of the identifiers used in the procedure (as in this example on Card 3) and it must contain a RETURN statement. The RETURN statement serves the same purpose in a procedure as in a subroutine (see Art. 2.9, p. 53).

The formal-parameter list has three subdivisions that are separated by semicolons. Starting at the left, the first subdivision contains a list of formal value-parameters that are separated by commas; e.g., RMAX, CMAX, RVMAX. The second subdivision contains a list of formal name-parameters that are separated by commas; e.g., M(,), V(), P(). The third subdivision contains formal program-reference parameters; e.g., ERR. The subdivisions are separated by semicolons.*

The value-parameters are variables which will be assigned a single value when the procedure is called. Hence, they permit only one-way communication, namely, from the program to the procedure. The name-parameters are identifiers of variables or arrays and will be replaced by corresponding identifiers from the main program at the time the procedure is called. This replacement results in a two-way communication between the program and the procedure, which is effective for the duration of procedure execution. A name-parameter that identifies an array comprises an identifier followed by parentheses. The parentheses contain one less comma than the dimension of the array; for example, M(,) and V() identify two and one-dimensional arrays, respectively. No subscript limits are specified and no ARRAY declarations for these quantities are required within the PROCEDURE because the actual arrays are defined and contained in the program outside the procedure. The program-reference parameters are identifiers representing statement labels; identifiers for subroutines, input-data sets, output-data sets, or formats; or identifiers for functions or procedures (in which case the identifier is followed by empty parentheses).

Execution of a procedure is initiated by a procedure-call statement that cannot precede the declaration for the procedure being called. As an example of a procedure-call statement, consider the following statement which calls for the execution of the procedure MATVECMULT:

```
MATVECMULT(I,J,K; A(,),B(),C(); LABEL1)
```

The procedure-call statement contains the name of the procedure and the actual-parameter list. The latter is enclosed in parentheses and must correspond element by element with the formal-parameter list of the procedure declaration. Thus, the actual-parameter list also contains three subdivisions: value-parameters, name-parameters, and

---

* In the reference manual Burroughs Algebraic Compiler, the three subdivisions are called input, output, and program-reference parameter lists, respectively. The terminology used here is introduced because it is descriptive of the functions of the lists and is consonant with the ALGOL terminology.

program-reference parameters. These subdivisions are separated by semicolons.

The list of actual value-parameters is a list of expressions, of which variables and numbers are special cases. The expressions are defined in terms of the identifier notation of the main program. It will be recalled that evaluation of an expression results in a single value. When the procedure-call statement is executed, each expression in the actual value-parameter list is evaluated, and the current value for each expression is assigned to its corresponding formal value-parameter prior to execution of the body of the procedure. Thus, in this example the current values of I, J, and K are assigned to RMAX, CMAX, and RVMAX.

The list of actual name-parameters should be a list of variables or array identifiers which are defined in the program outside the procedure. When the procedure-call statement is executed, each formal name-parameter is replaced by its counterpart actual name-parameter prior to the execution of the body of the procedure. The consequence of name replacement is that the procedure can operate directly on the variables and arrays defined in the program outside the procedure. Thus, a two-way channel of communication is set up between the program and the procedure.

Array identifiers in the actual name-parameter list must agree as to dimension with their counterparts in the formal name-parameter list. This means that the number of empty subscript positions must be identical in counterpart identifiers. In this example the dimensionality is identical for the arrays referred to in the procedure and their counterparts in the main program, hence the forms of the counterpart array identifiers in the formal and actual name-parameter lists are strictly analogous.

The dimension of an array in the main program may exceed that of the counterpart array referred to in the procedure. The resultant modification in the form of the array identifier in the actual name-parameter list is illustrated in a later example in this article.

The entities represented by the actual program-reference parameters must correspond to that of the formal program-reference parameters and they must be defined appropriately in the main program. Actual program-reference parameters may be statement labels; identifiers for subroutines, input-data sets, output-data sets, or formats; or identifiers of functions or procedures. In the last cases the identifier is followed by empty parentheses. An exception is that an intrinsic function may not be used as an actual program-reference parameter. If it is desired to do so, the function should be renamed by means of a FUNCTION declaration, after which the name of the function (followed by parentheses) can be used as an actual program-reference parameter. When the procedure-call statement is executed, the formal program-reference parameters are replaced by the actual program-reference parameters prior to execution of the statements in the procedure body. In the preceding example, the label LABEL1 will replace

ERR when the procedure-call statement is executed. Thus, whenever the GO TO ERR statement (Card 4) is executed within the procedure, the effect will be that of executing the statement GO TO LABEL1. Hence, there must be a statement in the main program prefixed with the label LABEL1.

For the most part, the body of a procedure is prepared in the same manner as a program. Some restrictions concerning procedures are the following: (1) The procedure body must be enclosed within BEGIN ... END brackets. (2) The body must contain a RETURN statement and must not contain a FINISH statement. (3) A PROCEDURE declaration may not contain a PROCEDURE declaration but may contain SUBROUTINE and FUNCTION declarations, and may refer to other procedures by means of program-reference parameters. (4) A procedure must be declared before it is called. (5) After a procedure is declared, the identifier which names it is recognized as such throughout the remainder of the program and it becomes (in effect) a reserved word for any program containing that procedure. However, the name of the procedure could be used as a formal parameter for another procedure or function declaration.

As another example of a procedure-call statement, consider the following portion of a program which defines the premultiplication of a matrix B by a matrix A, yielding a product matrix C = (AB). It is assumed that matrix A has m rows and n columns and matrix B has p rows and q columns. Furthermore, it is assumed that the numbers defining the sizes of the two matrices have been stored as elements of an array L:

$$L = \begin{bmatrix} m & n \\ p & q \end{bmatrix}$$

Thus, it is seen that m = L(1,1), n = L(1,2), etc. The execution of the following FOR statement would determine the product matrix C:

```
FOR J = (1,1,L(2,2));

    MATVECMULT(L(1,1),L(1,2),L(2,1);A(,),B(,J),C(,J); LABEL1);
```

The formal name-parameter list in the procedure declaration MATVECMULT contains the array identifiers M(,), V(), P(). The actual name-parameter list in the foregoing procedure-call statement is A(,), B(,J), C(,J). The form of M(,) is identical to A(,), as in the preceding example. The form of V() is not identical to that of B(,J) but the number of empty subscript positions is the same for both.

This is a case in which the dimension of the array in the main program exceeds that of the counterpart array referred to in the procedure declaration. In such a case, the dimensionality of actual and formal name-parameters are made to agree by fixing one (or more) of the subscripts in the actual name-parameter. This has the effect of temporarily reducing the dimension of the array while the procedure is executed.

In this example the counterpart of array V() is the jth column of B(,) and the counterpart of P() is the jth column of C(,). Thus the dimensions of the counterpart actual and formal name-parameters are identical for purposes of procedure execution.

It is not uncommon to find that a procedure is declared in which one or more of the parameter lists is empty. For example, the procedure heading may contain value and name-parameters, but no program-reference parameters. In such a case the semicolon which normally precedes the program-reference parameter list should be omitted. The mandatory use of semicolons in the procedure heading is shown by the following forms, in which VP, NP, and PRP represent lists of value-parameters, name-parameters, and program-reference parameters, respectively:

(VP;NP;PRP),(;NP;PRP),(VP;;PRP),(VP;NP),(;;PRP),(;NP),(VP)

The fifth form explains the requirement for semicolons in the READ and WRITE statements, which are actually procedure-call statements (see Arts. 3.7 and 3.8).

A procedure such as MATVECMULT, which is written in the BALGOL language, is known as a declared procedure. In this case, it is a procedure which is written by the programer himself, punched on cards, and then embodied in the main program. Another source of declared procedures is the computation center library, in which case the procedure would probably have been written by another programer. A deck of cards containing the procedure (in BALGOL) is obtained from the library and is inserted at an appropriate location in the main program by the programer, just as though he had written the procedure himself.

Other types of procedures are the previously-mentioned library procedures, which are discussed in Art. 2.14, and external procedures (see Art. 2.15).

2.13 Functions Defined by Procedures

Frequently the computations required to evaluate a mathematical function cannot be defined by a single arithmetic expression, and hence cannot be defined in a FUNCTION declaration. Instead, a PROCEDURE declaration is required to define the computational process. In such a case it is desirable that the procedure behave like a function in order that a procedure-call statement can be included in an expression as though it were a function designator.

For the procedure to behave like a function its execution must result in a single value that can be used in the evaluation of any expression containing it. This is accomplished by including in the body of the procedure a procedure-assignment statement that assigns a single value to the procedure identifier. The statement has the form

ID( ) = E

in which ID is the procedure identifier and E is an expression. The effect of this statement is to assign to the procedure identifier ID the current value of E. The procedure identifier must be followed by empty parentheses. It is mandatory that the procedure-assignment statement be followed immediately by a RETURN statement, if the procedure is to behave like a function.

As an illustration of such a procedure, let us consider the following PROCEDURE declaration which describes the computation of the series approximation to the exponential function:*

$$e^x = 1 + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

The identifiers used in the declaration have the following significance: XABS represents the absolute value of X, N represents the number of the term in the series, TERM represents the value of the Nth term, and SUM represents the value of the sum of the first N terms.

The PROCEDURE declaration is as follows:

```
                                                          Card No.
PROCEDURE EXPFCT(X);                                          1
   BEGIN XABS = ABS(X);                                       2
      IF XABS LSS 112.82666;                                  3
         BEGIN                                                4
            TERM = SUM = N = 1.0;                             5
            UNTIL (TERM/SUM) LEQ 0.00000001;                  6
               BEGIN                                          7
                  TERM = TERM(XABS/N);                        8
                  SUM = SUM + TERM;                           9
                  N = N + 1.0;                               10
               END;                                          11
            IF X LSS 0; SUM = 1.0/SUM;                       12
            EXPFCT( ) = SUM; RETURN                          13
         END;                                                14
      WRITE (;; ØDS, FØRM);                                  15
         ØUTPUT ØDS (X);                                     16
         FØRMAT FØRM (*X = *, X15.6,B1, *IS ØUT ØF RANGE
                      FØR EXPFCT*, W);                       17
      RETURN                                                 18
   END;                                                      19
```

* The library procedure EXP (see Art. 2.14) employs a rational function that can be computed more rapidly than the series approximation of e^x.

Comments on Procedure EXPFCT(X):

Card 1. The PROCEDURE heading defines the name of the procedure (EXPFCT) and the formal-parameter list. In this case the list consists of one formal value-parameter, namely, X. As a consequence, no semicolons appear in the formal parameter list, as explained on p. 64.

Card 2. The procedure body appears on cards 2 to 19 and is enclosed by the brackets BEGIN ... END.

Card 3. Since $e^{112.82666} \geq 10^{50}$, e^x is computed only when $|x| < 112.82666$. Otherwise, the program skips to the WRITE statement on Card 15 and an error message is printed.

Card 5. Initially, TERM, SUM, and N are set equal to one, which represents their values when the first term in the series is considered.

Cards 6 to 11. When this UNTIL statement is executed, the value of e^x is calculated to the accuracy specified by the UNTIL clause and the result is assigned to the variable SUM, representing the sum of N terms.

Card 12. If X is negative, then $e^x = 1/\text{SUM}$.

Card 13. Procedure-assignment statement. This statement assigns the value of SUM to EXPFCT( ). A RETURN statement follows immediately, as is required when a procedure is to behave like a function.

Card 15. This WRITE statement provides for printing an error message whenever XABS is outside the permitted range. It is followed by a RETURN statement (Card 18) in order to return to the main program.

Card 17. The FORMAT declaration (which is assumed to be punched in one card) provides for printing an output line in the following form:

X = ddddddd.dddddd IS OUT OF RANGE FOR EXPFCT

No type declarations appear in this procedure, hence all variables are implicitly floating.

A function defined as a procedure can appear directly in an expression, for example, the assignment statement

CF = EXPFCT(LAMBDA)

causes the above procedure to be executed with the current value of LAMBDA assigned to X. Then the resulting value of SUM (equal to EXPFCT(X)) is assigned to the variable CF.

The cards containing the function defined as a procedure are incorporated in the program deck in the same manner as a declared procedure.

2.14 Library Procedures

The BALGOL language includes a number of library procedures which can be called without the procedure being declared in the program. The general form of the call statement is the same for each of these procedures, i.e., the procedure name followed by the argument (a number, a variable, or an expression) which is enclosed within parentheses, thus:

```
NAME(E)
```

where NAME is the reserved word used as the name of the procedure. These procedure-call statements are included in arithmetic or Boolean expressions as required, in the same manner as described for intrinsic functions (see Art. 2.11).

Most of the library procedures are summarized in Table 2. In each case tabulated, the required type of the argument E and the type of the result are given, as well as the value which results from execution of the procedure. For more detailed information pertaining to these and other library procedures, the reader should refer to the Burroughs manual, Burroughs Algebraic Compiler.

2.15 External Procedures

An external procedure is a machine-language program that is stored in a deck of punched cards. When an external procedure is to be referred to in a program, it must be declared in the program and its deck of cards must be appended to the BALGOL program deck (see Art. 3.10). The declaration of the procedure must precede any other reference to the procedure.

The declaration for an external procedure has the following form:

```
EXTERNAL PROCEDURE ID (VP1,VP2,...; NP1,NP2,...; PRP1,PRP2,...)
```

where VP, NP, and PRP indicate value, name and program-reference parameters, respectively. It is seen from this example that the declaration consists of the declarator EXTERNAL PROCEDURE followed by a procedure heading of the same form as the procedure heading in a declared procedure (see Art. 2.12). The identifier ID and the list of parameters would be defined in a description of the external procedure obtainable from the computation center librarian.

Execution of an external procedure is called in the same manner as a declared procedure.

When the external procedure is to behave like a function, then the external procedure declaration must be followed by a type declaration for the procedure identifier ID, thus:

```
EXTERNAL PROCEDURE ID(parameter list); type declarator ID
```

A semicolon must be the only symbol (other than spaces) between the two declarations.

The external statement is a special case of the external procedure where the parameter list is empty. The primary function of the external statement is to provide a means for inserting a sequence of machine-language instructions (written by the programer) into a BALGOL program. Since machine language is beyond the scope of this text, external statements will not be discussed further.

TABLE 2. LIBRARY PROCEDURES

NAME ID	TYPE OF ARGUMENT E	VALUE OF RESULT	TYPE OF RESULT
SQRT	Floating	$\sqrt{E}$	Floating
EXP	Floating	e^E	Floating
LOG	Floating	ln E	Floating
SIN	Floating (E in radians)	sin E	Floating
COS	Floating (E in radians)	cos E	Floating
TAN	Floating (E in radians)	tan E	Floating
ARCSIN	Floating	arcsin E (radians)	Floating
ARCCOS	Floating	arccos E (radians)	Floating
ARCTAN	Floating	arctan E (radians)	Floating
SINH	Floating (E in radians)	sinh E	Floating
COSH	Floating (E in radians)	cosh E	Floating
TANH	Floating (E in radians)	tanh E	Floating
ENTIRE	Floating	Largest integer not greater than E	Floating
ROMXX	Floating	$\sqrt{1 - E^2}$	Floating
FLOAT	Integer	E	Floating
FIX	Floating	E	Integer

CHAPTER 3

ADDITIONAL TOPICS AND SUMMARIES

The articles in this chapter contain additional information pertaining to the BALGOL language, as well as summaries of features of the language which have been described previously. Also included is a list of reserved words, and a description of the organization of the assembled deck of cards that is to be processed by the computer.

3.1 Arithmetic Expressions

The concept of an arithmetic expression in BALGOL corresponds to that of an algebraic expression in algebra. Each is formulated in terms of constants, variables, arithmetic operators, and parentheses. A constant is a number which may have any of the three forms (integer, decimal, or floating point) defined in Art. 2.3. The concept of a variable in BALGOL is that of a variable in algebra. Thus a variable is an identifier that represents any unspecified value from a set of values.

The permissible arithmetic operators are illustrated in the following table of simple arithmetic expressions, in which V_1 and V_2 represent variables.

Simple Arithmetic Expressions in BALGOL	Meaning of Arithmetic Operator
$V_1 * V_2$	exponentiation
$V_1 . V_2$	multiplication
V_1 / V_2	division
$V_1 + V_2$, $V_1 - V_2$	addition, subtraction

It is permissible, of course, that V_1 and V_2 may be replaced by constants. However, it should be noted that the multiplication operator is identical to the decimal point. Hence, when V_1 and V_2 are both replaced by constants, ambiguity is avoided by requiring that one or the other of the two constants be enclosed in parentheses.

In the cases illustrated in the following table, the multiplication operator is redundant and may be omitted. The following notation is used in the table: ID is an identifier of a simple variable, an

array, or a function; V is an identifier of a simple variable; and C is a constant.

Form With Multiplication Operator	Equivalent Form With Multiplication Operator Omitted
).ID	)ID
).C	)C
).(	)(
V.(	V(
C.(	C(
C.ID	CID

Exponentiation (V_1*V_2) is undefined when $V_1 = 0$ and $V_2 \leq 0$, or when $V_1 < 0$ and V_2 is nonintegral. Otherwise the values of V_1 and V_2 may be positive or negative, and integer or noninteger.

Division (V_1/V_2) is undefined when $V_2 = 0$.

It is permissible to replace V_1 and V_2 (in the above table of simple arithmetic expressions) with function designators (see Art. 2.10) or by procedure-call statements when the procedures behave like functions (see Art. 2.13).

More complicated arithmetic expressions may be formed by composition of simple arithmetic expressions. Thus it is permissible to replace the variables V_1 and V_2 in the simple arithmetic expressions given above by expressions E_1 and E_2 of arbitrary complexity.

Parentheses should be used freely by the programer, just as in algebra, to obviate ambiguity with respect to the order in which operations are performed in the evaluation of an expression. For example, A/B/C should be either (A/B) or A/(B/C), and A*B*C should be either (A*B)*C or A*(B*C). Within parentheses, the order of precedence of performing operations is exponentiation, multiplication, division, then addition or subtraction. Subject to these restrictions operations are performed working from right to left in an expression.

Note that each arithmetic operator requires two operands. The type of arithmetic in which an operation is performed depends upon the types of the values of the operands. Each operand must, of course, represent a single numeric value. An operation will be performed in integer arithmetic if and only if both operands are values of type integer, and the resulting value will also be type integer. In integer arithmetic, division always yields an integer quotient; e.g., 8/3 results in the quotient 2, and 4/6 results in the quotient 0. When the magnitude of the product of two integers exceeds 9999999999, the result is truncated from the left and only the low order digit positions are retained.

If either or both of the operands are values of type floating, then the operation will be performed in floating-point arithmetic. Any necessary transformations from type integer to type floating will be performed automatically prior to execution of the operation, and the resulting value will be type floating.

Similarly, when the value resulting from evaluation of an arithmetic expression is assigned to a variable of a different arithmetic type, the necessary transformation in type is performed automatically prior to execution of the assignment operation.

It is permissible to replace V_1 and V_2 in an arithmetic expression with Boolean expressions (see next article). Evaluation of a Boolean expression yields either the value 1 or 0, and when these values are encountered in an arithmetic expression they are treated in every respect as integers.

3.2 Boolean Expressions

The use of a Boolean expression to specify a condition in an IF clause has been illustrated in Art. 2.2 (see p. 22). However, Boolean expressions have a much wider range of application than is indicated in that illustration. For example, they are used in problems involving symbolic logic, Boolean algebra, switching circuits, etc.

A Boolean expression is formulated in terms of Boolean constants, Boolean variables, Boolean operators, arithmetic expressions, relational operators, and parentheses.

The Boolean constants are the digits 1 (representing true) and 0 (representing false). Boolean variables are named by means of identifiers in the same manner as arithmetic variables. Since the rules for formulating identifiers are independent of the properties of the entity identified, it is necessary to declare Boolean variables as to type. This is accomplished in a manner that is analogous to declarations for variables of type integer and type floating, the difference being that the declarator in this case is the reserved word BOOLEAN. As an example, the following declaration might be used:

```
BOOLEAN S, ABLE, BAKER, B1, B2
```

where S, ABLE, BAKER, B1, and B2 are identifiers of variables. A Boolean variable may be either single valued or array valued. In the latter case, the size and dimensionality of the array is defined in an array declaration in the manner previously described (see Art. 2.6, p. 37).

The relational operators are a special set of Boolean operators for which the operands may be arithmetic expressions. The relations

BALGOL	Mathematical Equivalent
E_1 LSS E_2	$E_1 < E_2$
E_1 LEQ E_2	$E_1 \leq E_2$
E_1 EQL E_2	$E_1 = E_2$
E_1 NEQ E_2	$E_1 \neq E_2$
E_1 GEQ E_2	$E_1 \geq E_2$
E_1 GTR E_2	$E_1 > E_2$

TABLE 3. RELATIONAL OPERATORS

shown in Table 3 can be specified in BALGOL. The symbols E_1 and E_2 represent arithmetic expressions. The operators LSS, LEQ, etc., are called relational operators and must be preceded and followed by a space. The meaning of each operator is indicated by its familiar counterpart shown in the column headed mathematical equivalent. Note that the mathematical symbol for equality is used in BALGOL as the assignment operator, whereas the remaining mathematical symbols shown above are undefined in BALGOL.

As illustrated earlier in Arts. 2.1 and 2.7, a simple relation may appear in an IF clause or an UNTIL clause without being enclosed in parentheses. For example, the clause IF E1 LSS E2 is a well-defined IF clause. However, when a relation is to be evaluated as a Boolean expression, it must be enclosed in parentheses. Thus, V = (E1 LSS E2) is a well-defined assignment statement, in which (E1 LSS E2) is a Boolean expression. Normally V would have been declared previously as type Boolean, and it would be assigned the value 1 (true) if the value of E1 is algebraically less than E2; otherwise, it would be assigned the value 0 (false).

The remaining Boolean operators are NOT, AND, OR, IMPL, and EQIV. The meaning of each operator is defined below in terms of the values that result from its application. It is assumed below that B1 and B2 are Boolean variables; however, the meanings hold when either or both are replaced by Boolean expressions.

The Boolean expression NOT B1 has the value 1 when B1 has the value 0, and NOT B1 has the value 0 when B1 has the value 1. Note that the operator NOT has only one operand which always follows and never precedes the operator.

The expression B1 AND B2 has the value 1 if and only if B1 and B2 simultaneously have the value 1. Otherwise the expression B1 AND B2 has the value 0.

The expression B1 OR B2 has the value 0 if and only if both B1 and B2 simultaneously have the value 0. Otherwise it has the value 1.

The expression B1 IMPL B2 has the value 0 if and only if B1 has the value 1 while B2 simultaneously has the value 0. Otherwise it has the value 1.

The expression B1 EQIV B2 has the value 1 if and only if B1 and B2 simultaneously have identical values. Otherwise it has the value 0.

The meanings of the Boolean operators are summarized in Table 4, which in symbolic logic is referred to as a truth table. The table shows the values assumed by each expression when the variables B1 and B2 assume each of the four possible pairs of values.

B1	0	0	1	1
B2	0	1	0	1
NOT B1	1	1	0	0
B1 AND B2	0	0	0	1
B1 OR B2	0	1	1	1
B1 IMPL B2	1	1	0	1
B1 EQIV B2	1	0	0	1

TABLE 4. TRUTH TABLE

It was stated earlier that B1 and B2 could represent Boolean expressions as well as variables. This will be illustrated in the following example.

Consider the problem of adding two binary integers. This will involve an addend A1, an augend A2, a sequence of "carries" C, and the resultant sum S, as shown by the example in Fig. 6.

In order to describe the generation of the sum S in a BALGOL program in terms of Boolean expressions, let J be an index over the digit positions of the numbers shown in Fig. 6. Thus C, A1, A2, and S each become array-valued Boolean variables, and the first element in each array is the units position of the corresponding binary integer, the second element is the 2's position, etc.

Now consider the partial sum, PS(J) and the contribution PC(J) to C(J+1) that results from adding the pair of elements A1(J) and A2(J). The truth table shown in Fig. 7 shows the outcome for each possible pair of values of A1(J) and A2(J).

Carries	C	0	1	1	0
Addend	A1	0	0	1	1
Augend	A2	1	0	1	1
Sum	S	1	1	1	0

FIG. 6

A1(J)	0	0	1	1
A2(J)	0	1	0	1
PS(J)	0	1	1	0
PC(J)	0	0	0	1

FIG. 7

Referring back to the truth table that summarizes the meanings of the Boolean expressions, it is seen that the assignment statement

PS(J) = NOT (A1(J) EQIV A2(J))

results in PS(J) being assigned the appropriate value for each pair of values that A1(J) and A2(J) can assume. Similarly, it is seen that the statement

PC(J) = A1(J) AND A2(J)

results in PC(J) being assigned the appropriate values.

C(J)	0	0	1	1
PS(J)	0	1	0	1
S(J)	0	1	1	0
PC2(J)	0	0	0	1

To complete the calculation of S(J) it is necessary to add C(J) to PS(J) as shown in the accompanying table. The variable PC2(J) represents the resultant contribution to C(J+1).

Thus the following assignment statements serve to calculate the values of S(J) and PC2(J):

```
S(J) = NOT (C(J) EQIV PS(J))
PC2(J) = C(J) AND PS(J)
```

Finally, the value of the carry to position J+1 is calculated by the assignment statement

```
C(J+1) = PC(J) OR PC2(J)
```

Alternatively, the values of S(J) and C(J+1) could be calculated as follows:

```
PS(J) = NOT (A1(J) EQIV A2(J))
S(J) = NOT (C(J) EQIV PS(J))
C(J+1) = (A1(J) AND A2(J)) OR (C(J) AND PS(J))
```

Note that, as in the case of arithmetic expressions, parentheses are used freely to obviate ambiguity. Within parentheses the order of precedence of executing Boolean operations is NOT, AND, OR, IMPL, EQIV.

3.3 The GO TO and SWITCH Statements

The normal sequence of execution of statements in a BALGOL program is in the order of their appearance. It has been demonstrated previously, however, that the normal sequence may be varied by the execution of a GO TO statement which has the form

```
GO TO SL
```

where SL represents a statement label. When a GO TO statement is executed, its successor is that statement which is prefixed with the label SL.

The SWITCH statement may be regarded as a generalization of the GO TO statement. The SWITCH statement has the form

SWITCH E, (L_1, L_2, ... , L_k)

where E represents an arithmetic expression; L_1, L_2, ... , and L_k represent statement labels; and k represents the number of labels in the list.

When the SWITCH statement is executed, the expression E' = |E| is evaluated. If E' is of type floating, it is automatically truncated to an integer i; otherwise, i = E'. The integer i serves as an index for selecting the label of the successor to the SWITCH statement, according to the following scheme:

If i = 0, no label is selected and the normal sequence of statement execution is unaffected.

If $1 \leq i \leq k$, the i-th label, counting from left to right, in the list is selected; thereby identifying the successor to the SWITCH statement.

If $i > k$, the successor to the SWITCH statement is undefined, and the consequences cannot be predicted.

3.4 EITHER IF Statement

The EITHER IF statement is basically a sequence of IF statements and hence may be considered as a generalization of the IF statement, which has been discussed previously in Arts. 2.1 and 2.2. In summary, it consists of an IF clause followed by a statement. The IF clause specifies a condition, and the statement following the clause is executed if and only if the condition is satisfied.

The EITHER IF statement has two alternative forms, the difference being in the form of the ending of the statement:

EITHER IF B_1; S_1; OR IF B_2; S_2; ... ; OR IF B_k; S_k END

EITHER IF B_1; S_1; OR IF B_2; S_2; ... ; OTHERWISE; S_k

In both cases each B_i represents either a Boolean expression of arbitrary complexity or a relation, and each S_i represents either a simple statement or a compound statement. The EITHER IF ... OTHERWISE form of the statement is illustrated in the example of Art. 2.7 (see p. 44).

When the EITHER IF ... END form of the statement is executed, the Boolean expressions B_1, B_2, etc., are evaluated successively until one is found that has the value true (that is, 1). Then the statement immediately following the clause containing that expression is executed. Execution of that statement may result in execution of a GO TO statement that specifies its successor statement; however, in any case execution of the EITHER IF statement is considered to be completed. If no Boolean expression is found that has the value true, then none of the statements S_1, S_2, ... , S_k is executed, and statement execution continues in the normal sequence.

The EITHER IF ... OTHERWISE form of the EITHER IF statement differs from that of the EITHER IF ... END form in that the OTHERWISE clause has replaced the k-th IF clause. The OTHERWISE clause behaves exactly like an IF clause containing a Boolean expression which has the constant value true. Thus, whenever the sequence of expressions

B_1, B_2, ... , B_{k-1} all have the value false (0), the statement S_k is executed.

It is not necessary that the EITHER IF ... OTHERWISE form shall contain any OR IF clauses. When it does not, one or the other of two statements is executed depending upon the value of the Boolean expression.

3.5 FOR Statement

The use of a FOR statement was illustrated previously in Art. 2.6 (see p. 38) and the discussion given there is sufficient for the majority of programing purposes. However, in this article a comprehensive discussion of the statement is presented.

The general form of the FOR statement is

FOR V = IL; S

where FOR is a reserved word, V is any variable (called the induction variable), IL is an iteration list containing arithmetic expressions and/or step-until elements, and S is a statement that may be a simple statement, a compound statement, a conditional statement, etc. The FOR clause assigns successive values from the iteration list to the variable V. After each assignment, the statement S is executed. This cycle is repeated until either the iteration list is exhausted, at which point it is recognized that execution of the FOR statement has been completed, or a GO TO statement contained in S is executed and the cycle is thereby interrupted.

The iteration list may contain a sequence of arithmetic expressions* separated by commas, e.g., E_1, E_2, ... , E_n. Furthermore, a combination of elements of this type and one or more step-until elements may be used in the iteration list, e.g., E_1, E_2, (E_I, E_S, E_T), E_3,...,E_n. In other words, any arbitrary combination of expressions and step-until elements may be used.

The general form of the step-until element is

(E_I, E_S, E_T)

where E_I, E_S, and E_T are arithmetic expressions that define, respectively, the initial value of the generated sequence, the increment or "step" between successive values, and the test or limiting value. The expressions E_I, E_S, and E_T may assume either positive or negative values. When the sequence passes the test value, the step-until element is deemed to be exhausted.

---

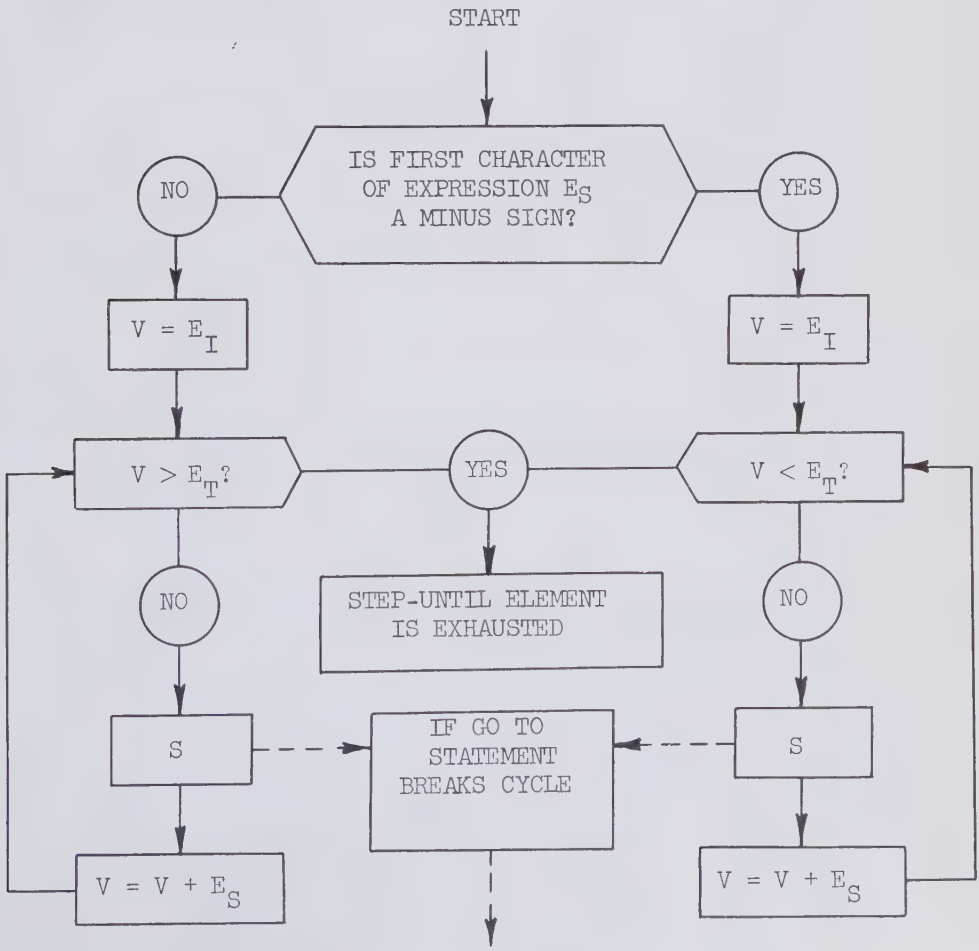
* Throughout this discussion it should be recalled that numbers and variables are special cases of arithmetic expressions.

There are two forms of the step-until element. The following convention has been adopted to distinguish between these two forms. If the first character in the expression E_S is a minus sign, for example, -6 or -Y*2+B, then the step-until element will be called a "step-down" element. Otherwise, it will be called a "step-up" element.

In the case of the step-down element, it is necessary that $E_I > E_T$ algebraically, and the element is exhausted when the value of the induction variable V becomes less (algebraically) then the value of E_T. In the step-up element, it is necessary that $E_I < E_T$ algebraically, and the element is exhausted when the value of V exceeds (algebraically) that of E_T.

The flow chart shown below illustrates the functioning of a FOR statement while stepping through a step-until element.

Flow Chart for Step-Until Element:



The expression E_S is evaluated each time the induction variable V is incremented and the expression E_T is evaluated each time it is compared with V. Thus the increment between steps and the limiting value can be varied from step to step. For example, the values of E_S and E_T can be a function of values calculated by the statement following the FOR clause.

The only difference between the functioning of the step-up and step-down elements is in the comparison that determines when the step-until element is exhausted, i.e., $V > E_T$, and $V < E_T$, respectively. If a step-up element is to be strictly increasing, it is necessary that $E_S > 0$. If a step-down element is to be strictly decreasing, it is necessary that $E_S < 0$. Note that the minus sign which is used to indicate the step-down element is part of the expression E_S and hence is taken into account in evaluating E_S.

Some examples of FOR clauses are as follows:

```
FOR V = 2, 3, 11, 6, 18                    FOR V = A, B, A*2-B, C
FOR V = (1,1,9), (9.1,0.1,10)              FOR V = (C, D, E)
FOR V = -25, (-20,1,20), 25                FOR V = (X*2, -1, 3X)
FOR V = (10.0, -0.1, 0)                        etc.
```

These examples should be sufficient to indicate the flexibility that is available to the programer in constructing a FOR clause for use in a FOR statement.

3.6 Filling an Array

The construction of ARRAY declarations was discussed previously in connection with the illustrative example of Art. 2.6 (see p. 37). In that example, values were assigned to elements of the array by means of assignment statements within the program, and the ARRAY declaration itself served to denote the dimension and size of the array.

An alternative form of the ARRAY declaration permits the programer to specify in the declaration a sequence of values that are to be assigned to the elements of the array at the time the program is compiled. For example, the declaration

```
ARRAY X(2,3) = (1, 5.0, 4, 6.0)
```

would result in the integer values 1, 5, and 4 being assigned to the elements X(1,1), X(1,2), and X(1,3), respectively, and the values 6, 0, and 0 being assigned to the elements X(2,1), X(2,2) and X(2,3), respectively, provided that X had been declared as type integer. If instead X had been declared as type floating, then each value would be transformed automatically to the excess-fifty floating point form

as the assignments were made. Note that only four values are specified in the list, whereas there are six elements in the array. When the list of values is exhausted, the remaining elements in the array are assigned automatically the value zero.

This alternative form of the ARRAY declaration may be generalized for a three-dimensional array as follows:

$$\text{ARRAY ID}(U_1, U_2, U_3) = (n_1, n_2, n_3, \ldots, n_m)$$

Where U_1, U_2, U_3 represent integers that specify the respective upper bounds of the corresponding subscripts of the subscripted variable $\text{ID}(E_1, E_2, E_3)$; n_1, n_2, ... represent numeric values; and E_1, E_2, and E_3 represent arithmetic expressions.

During compilation, the following assignment of the numerical values n_1, n_2, ... , n_m to the elements ID(I,J,K) of the array would be made automatically:

Sequence of Values	Sequence of Elements
The 1st U_3 values	$\text{ID}(1,1,1)$ to $\text{ID}(1,1,U_3)$
The 2nd U_3 values	$\text{ID}(1,2,1)$ to $\text{ID}(1,2,U_3)$
The 3rd U_3 values	$\text{ID}(1,3,1)$ to $\text{ID}(1,3,U_3)$
⋮	⋮
The last U_3 values	$\text{ID}(U_1,U_2,1)$ to $\text{ID}(U_1,U_2,U_3)$

It is assumed in the above list that the number m of numeric values is at least as great as $(U_1)(U_2)(U_3)$. In the event that the list of numeric values is exhausted before each element in the array is assigned a value, the remaining elements would be assigned automatically the value zero.

3.7 The READ Procedure

The use of the READ statement has been illustrated in several articles in Chapter 2. It is actually a statement which calls for execution of the library procedure named READ. Execution of the READ procedure causes one or more cards to be read and assigns the sequence of values thus read to a sequence of variables specified in the input-data set that is referred to in the READ statement.

The general form of the READ statement is

READ(; S; IDIDS)

where the name-parameter S represents a variable of type Boolean and

the program-reference parameter IDIDS is the identifier of an input-data set that is defined in an INPUT declaration. Inclusion of the Boolean variable in the READ statement is optional rather than mandatory. Its purpose is discussed later.

An INPUT declaration is identified by the declarator INPUT and has the general form

$$\text{INPUT IDIDS}_1\ (\text{IDS}_1),\ \text{IDIDS}_2\ (\text{IDS}_2),\ \ldots\ ,\ \text{IDIDS}_k\ (\text{IDS}_k)$$

where IDS represents an input-data set. There may be more than one INPUT declaration in a program. There is no restriction as to the relative locations within the program of an INPUT declaration that defines an input-data set and a READ statement that refers to that input-data set.

In its simplest form an input-data set is a list of variables, for example,

$$V_1,\ V_2,\ \ldots\ ,\ V_k$$

where each element in the list represents either a simple variable or a subscripted variable. In an INPUT declaration, the input-data set is enclosed in parentheses and preceded by an identifier, which may be referred to in a READ statement.

When a READ statement is executed, card reading is initiated. Subsequent events depend upon whether or not a Boolean variable appears in the READ statement. When there is no Boolean variable in the READ statement, successive values (reading left to right) from the card are assigned to successive variables (reading left to right) in the input-data set. If required, card reading is continued automatically until a value is assigned to each variable in the input-data set, at which point execution of the READ statement is complete. Thus, it is seen that any unassigned values on the last card read are ignored.

A variation of the basic form of the input-data set permits an iterated element which defines a sequence of subscripted variables; for example,

$$\text{FOR } V_1 = IL_1;\ \text{FOR } V_2 = IL_2;\ \ldots\ ;\ \text{FOR } V_k = IL_k;\ V(V_1,\ V_2,\ \ldots\ ,\ V_k)$$

In this form, V_1, V_2, ... , and V_k represent induction variables, IL_1, IL_2, ... , and IL_k represent iteration lists, and $V(V_1,V_2,\ldots,V_k)$ represents a subscripted variable. The behavior of the FOR clauses in this element is strictly analogous to their behavior in the FOR statement (see Art. 3.5). The values of variables appearing in the iteration list may be assigned within the program or alternatively they may be read from data cards. For example, consider the input-data set

$$K,P,\ \text{FOR I} = (K,1,P);\ X(I)$$

Since K and P will be assigned values before the FOR clause is executed, those variables can be used to control the limits of the step-until element in the FOR clause.

A variation of the iterated element (discussed above) permits replacement of the subscripted variable with an input-data set enclosed in parentheses, for example,

$$\text{FOR } V_1 = IL_1; \text{ FOR } V_2 = IL_2; \ \ldots \ ; \text{ FOR } V_k = IL_k; \ (IDS)$$

In this form, the FOR clauses control iteration of the input-data set represented by IDS.

When a Boolean variable is included in the READ statement, the operation of the READ procedure differs in the following respects from that described above. Each time a card is read, a check is made automatically to determine whether or not the word SENTINEL appears in card columns 2 to 9. If it does appear, the Boolean variable specified in the READ statement is automatically assigned the value 1, and execution of the READ statement is terminated immediately. Otherwise, the Boolean variable is automatically assigned the value 0 and the operation of the READ procedure assigns successive values from the cards to successive variables in the input-data set as described above. Thus a SENTINEL card may be used to indicate the end of a sequence of data cards.

The READ procedure is oblivious to the declared types of the variables in the input-data set. The form in which the assigned values are stored in memory is determined by the form in which they appear in the data card (see Art. 2.3). The programer is responsible for determining that the form of each value in a data card is consistent with the type of the variable to which it is to be assigned.

Individual numeric values on data cards must be separated from each other by at least one blank column, and each must be wholly contained within columns 2 to 80 of one data card. In addition, each must satisfy the appropriate restriction as to the number of significant digits as specified in Art. 2.3. Alphanumeric values in data cards are bracketed by semicolons to distinguish them from numeric values (see Art. 2.8).

3.8 The WRITE Procedure

The use of the WRITE statement has been illustrated in several articles in Chapter 2. It actually calls for execution of the library procedure named WRITE. This procedure provides three alternative forms of recording output information: printed pages, punched cards, and typed pages. The word "write" as used in this book encompasses all three media.

The WRITE procedure requires that the programer must specify three things: (1) an output-data set consisting of the sequence of values to be written; (2) an editing phrase for each value to be written, thereby specifying the form (integer, decimal, or floating-point) and

the width of the field in which the number is to be written; and (3) an activation phrase for each line that is to be written, thereby specifying the output medium. In addition, the WRITE procedure permits the programer to insert alphabetic and numeric information before, between, or after the values specified in the output-data set. This feature is provided by the alphanumeric-insertion phrase.

In general the WRITE statement has the form

WRITE (;; IDODS, IDFORMAT)

This is the form of a procedure-call statement where the value-parameter and the name-parameter lists are empty (see p. 64). The terms IDODS and IDFORMAT are used to indicate that the program-reference parameter list comprises an identifier for an output-data set, and an identifier for a format specification (hereafter called a format).

It is permissible for the output-data set to be vacuous, in which event the WRITE statement has the form

WRITE (;; IDFORMAT)

This form may be used for writing page headings, messages, etc., when all of the output information is specified in alphanumeric-insertion phrases.

The output-data set itself is defined in an OUTPUT declaration, which is identified by the declarator OUTPUT and contains one or more output-data sets, for example:

OUTPUT $IDODS_1(E_{11}, E_{12}, \ldots), IDODS_2(E_{21}, E_{22}, \ldots), \ldots$

Each output-data set is preceded by an identifier which serves as the name by which it is referred to in a WRITE statement. The output-data set itself is enclosed in parentheses and consists of a sequence of expressions separated by commas. When an expression is included in an output-data set, it is evaluated during execution of the WRITE procedure. When a variable is included in an output-data set, the value currently assigned to the variable is written by the procedure.

When an output-data set contains a multi-valued variable, one or more FOR clauses may be used to specify the sequence of subscripted variables, for example:

$IDODS(E_1, E_2, \ldots,$ FOR I = (1,1,P); FOR J = (1,1,Q); X(I,J), $E_n, E_{n+1}, \ldots)$

Note that the entire iterated element of the output-data set (consisting of the two FOR clauses and variable X(I,J)) is separated from each adjacent element by commas. The FOR clauses operate in their usual manner to control the sequence of values assumed by the subscripts of the variable X(I,J).

The format referred to (by means of its identifier) in the WRITE statement is defined in a FORMAT declaration which, in turn, is identified by the declarator FORMAT and contains one or more formats,

for example:

FORMAT IDFORMAT$_1$(format), IDFORMAT$_2$(format), ...

Each format is enclosed in parentheses and is preceded by its identifier.

A format for an output-data set such as IDODS(E_1,E_2,...,E_k) would provide an editing phrase for each variable (or expression) in the output-data set, and an activation phrase for each line to be printed, for example:

IDFORMAT(EP_1, EP_2, ... , AP_1, ... , EP_k, AP_h)

Where EP_1 is the editing phrase for value E_1, EP_2 for E_2, ... , and EP_k for E_k. Note that, in this example, successive values specified in the output-data set are paired with corresponding editing phrases. Thus the procedure operates from left to right in both lists, editing the sequence of values to be written on a single line. When the activation phrase AP_1 is encountered, the preceding edited line is written. The procedure repeats this editing process and the next sequence of edited values is written when the next activation phrase is encountered. The process continues until all values specified in the output-data set have been edited.

There are four types of editing phrases for numeric values. An editing phrase specifies the form (integer, decimal, or floating-point) and the width of the field (number of spaces) in which the value is to be written. In each of the following definitions of editing phrases w represents an integer number that specifies the width of the field, and d also represents an integer number. The value, when written, is always right justified in the field, and if the number will not fit in the field, an asterisk is written in lieu of the number.

Editing Phrases for Numeric Values:

Iw The letter I signifies that the value is to be written in integer form. The field width w must be great enough to accommodate the value and (in the case of a negative number) a sign.

Xw.d The letter X signifies that the value is to be written in decimal form, that is, containing a decimal point. The integer d specifies the number of decimal places to which the value is to be truncated when it is written. The field width w must be great enough to accommodate the number and its decimal point, plus a sign if the number is negative.

Sw.d The letter S signifies that the value is to be written in decimal form. The integer d specifies the total number of digits which will be written, including zeros following the decimal point if the number is less than 0.1. The field width w must be great enough to accommodate d digits, a decimal point, and a possible minus sign.

Fw.d The letter F signifies that the value is to be written in floating-point form. The integer d specifies the number of significant digits to which the number is to be truncated when it is written. In this type of editing phrase it is necessary that w be great enough to provide for a possible minus sign, a decimal point, d digits, a comma, and a scale factor (which may have a minus sign).

Each of the above phrases edits a single numeric value. The editing process assumes that the value is stored in memory in a form that is consistent with the specified editing phrase. The letter I assumes that the value is stored in integer form while the letters X, F, and S assume that the value is stored in excess-fifty floating point form. The WRITE procedure is oblivious to the declared types of variables (or expressions) in the output-data set, hence the programer must ensure that the form of each value to be written is consistent with its corresponding editing phrase.

The alphanumeric editing phrase Aw signifies that one or more values specified in the output-data set are to be edited as a string of w alphanumeric characters. Since a word in memory can store the decimal equivalents for five characters (see Art. 2.8), the alphanumeric editing phrase Aw corresponds to n successive variables in the output-data set, where n is the smallest integer not less than w/5. When w is not an even multiple of 5 the value of the last variable is truncated automatically.

The format IDFORMAT, shown above, can be modified by interspersing blank-insertion phrases and alphanumeric-insertion phrases. The blank-insertion phrase, Bw, in a format causes w spaces to be inserted in the edited line. The alphanumeric-insertion phrase is an arbitrary string of characters (the asterisk excluded) enclosed in bracketing asterisks. It causes the string of characters between the asterisks to be inserted in the edited line (see Art. 2.5, p. 34).

The editing phrases for numeric and alphanumeric values, and for the blank-insertion phrase can be converted to definite repeat phrases by prefixing the basic phrase with an integer that specifies the number of repetitions. For example 3I5 is equivalent to I5, I5, I5.

As stated earlier, an activation phrase causes the edited line to be written. The following activation phrases are commonly used, but additions and modifications to this list may exist at a particular computation center.

Activation Phrases:

Ws The letter W signifies that the edited line is to be written by the line printer. The letter s indicates an integer that is used (as defined below) to control spacing between lines.

P The letter P signifies that the edited line is to be punched in a card.

Cs The letter C signifies that the edited line is to be both written by the line printer and punched in a card. Only the first 80 characters in the edited line will be punched.

Tc The letter T signifies that the edited line is to be typed on the typewriter (supervisory printer). The value of c specifies the number of carriage returns to be executed prior to typing.

Values of s:

0 or Blank	Single space before printing
1	Eject page after printing
2	Single space before and after printing
3	Eject page before printing
4	Double space before printing

If one activation phrase follows another, the effect of the second one will be to cause writing of a blank line.

Two or more consecutive phrases in a format may be enclosed in parentheses to form a compound editing phrase. Thus a format specification itself is a compound editing phrase. Compound editing phrases may be nested in a manner analogous to the nesting of algebraic expressions. A compound editing phrase is automatically an indefinite-repeat phrase, i.e., it will be repeated until the output-data set is exhausted. Any compound phrase nested in the format specification may be converted to a definite-repeat phrase by prefixing it with an integer that specifies the number of repetitions.

There are two particular cases of editing that should be noted. If the number of values specified in the output-data set exceeds the effective number of editing phrases in the format specification, the entire format specification is repeated until the output-data set is exhausted. If, on the other hand, the output-data set is exhausted before the editing phrases are exhausted, the surplus editing phrases are ignored.

A line is limited to 120 spaces on the line printer and 80 spaces on a card. If an edited line exceeds the limit for the output medium specified in the activation phrase, the excess data is lost.

3.9 Error Messages

It was stated in Art. 2.1 that error messages are printed automatically during the compilation phase whenever certain errors in syntax are recognized by the compiler. Thus the error messages are interspersed among the lines of the program listing that is printed automatically as compilation proceeds. The message itself, e.g., CONSTANT OUT OF RANGE, indicates the nature of the error, while the location of the message indicates the relative location of the error in the program.

In addition, other error messages may be printed automatically when certain events transpire in the execution of a program. For example, the error message ARITHMETIC OVERFLOW indicates that in the execution of some statement an attempt has been made to calculate a result R which is greater (numerically) than the largest value that can be stored in memory. However, there are circumstances under which an overflow can occur without an error message being printed. Thus the absence of the ARITHMETIC OVERFLOW message does not guarantee the absence of overflow errors during program execution.

The precise meaning of a specific error message can be found in the reference manual Burroughs Algebraic Compiler.

3.10 Assembling the Card Deck for Processing

A card deck that is to be processed comprises at least a Job card, a Lodx card, and the BALGOL program cards. This deck is supplemented, as required, by one or more card decks containing data cards, declared procedures, and external procedures. If the desired results are to be obtained, it is imperative that the required decks shall enter the input unit of the computer in the following sequence: 1) the Job Card; 2) the Lodx Card; 3) if required, declared procedures obtained from the computation center or other sources; 4) the BALGOL program, which always concludes with a FINISH statement; 5) if required, a deck for each external procedure declared in the program (the last such deck must be followed by a card containing the word FINISH); and 6) if required, data cards.

3.11 Summary of Rules of Precedence for Declarations

The type of an identifier must be declared (explicitly or by default) before that identifier is used in any statement or in any other declaration.

The type of an identifier for an array must be declared (explicitly or by default) before the ARRAY declaration.

An array must be defined in an ARRAY declaration before the use of any variable with subscripts which represents an element of that array.

The declaration of a FUNCTION must precede any statement in which the function is called.

The declaration of a PROCEDURE must precede any statement in which the procedure is called.

A FINISH declaration followed by a semicolon must terminate the program.

3.12 List of Reserved Words

There are certain combinations of alphabetic characters that have a definite meaning within BALCOM and can not be used by the programer for any other purpose. These are referred to as reserved words and include declarators, operators, separators, the names of library procedures, intrinsic functions, etc. The following list of these words is complete as of the time that this text is written. As library procedures are developed to increase the versatility of BALGOL, it will be necessary to reserve additional words.

ABS	FINISH	MAX	SEGMENT
AND	FIX	MIN	SIGN
ARCCOS	FLOAT	MOD	SIN
ARCSIN	FLOATING	MONITOR	SINH
ARCTAN	FOR		SQRT
ARRAY	FORMAT	NEQ	STATEMENT
	FUNCTION	NOT	STOP
BEGIN			SUBROUTINE
BOOLEAN	GEQ	OR	SWITCH
	GO	OTHERWISE	
COMMENT	GTR	OUTPUT	TAN
COS		OVERLAY	TANH
COSH			TO
	IF	PCS	
EITHER	IMPL	PROCEDURE	UNTIL
END	INPUT		
ENTER	INTEGER	READ	WRITE
ENTIRE		REAL	
EQIV	LEQ	RETURN	
EQL	LOG	ROMXX	
ERROR	LSS		
EXP			
EXTERNAL			

Most of the words in the above list are explained elsewhere in the text. For a discussion of the remaining words such as ERROR, MONITOR, OVERLAY, and SEGMENT, the reader should refer to the reference manual Burroughs Algebraic Compiler.

INDEX