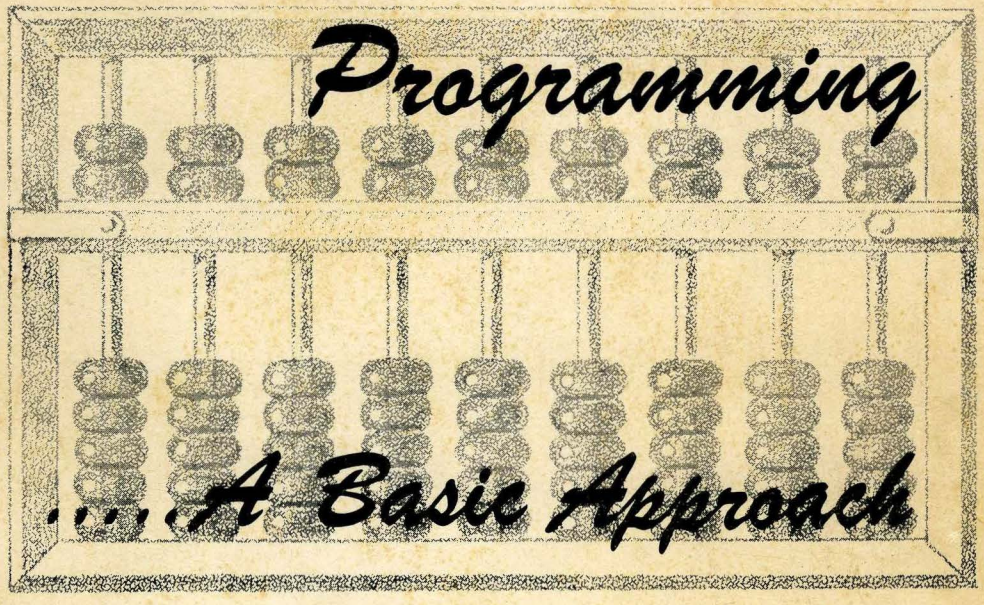
ALGOL



THURNAU . JOHNSON . HAM

ALGOL PROGRAMMING

A BASIC APPROACH

Although Burroughs Extended ALGOL contains many constructs not found in ALGOL 60, the subset covered in this text contains very few of them. With the exception of the character set, nearly every ALGOL 60 construct is contained in Extended ALGOL. Thus, this text should serve as a good introduction to other reasonably pure implementations of ALGOL 60. Appendix D contains a discussion of the differences which exist between the language covered herein and ALGOL 60 as defined in the ALGOL report.

Our objective has been the creation of a textbook on ALGOL rather than a self-teaching document or a reference manual. Therefore, we would expect this material to be most useful in a normal course environment or to the individual who already has a computing background and is interested in learning ALGOL. Certain knowledge, such as the preparation of punched card decks, etc., is assumed and should be provided to the novice by an instructor or obtained from other textual material. Although the text is not primarily a reference manual, its many definitions and examples make it useful as such to the programmer.

The authors would like to express their sincere thanks to Mr. A. Y. Wilson of the Burroughs Corporation whose confidence and support made the writing of the text possible. Our thanks also to W. H. Eichelberger, Major W. D. Marsland, Warren Gaynor, Darrell Albee, and A. H. Rabenau who made valuable suggestions for and provided critical comment on the material herein. Finally, many thanks to Miss Kay Shackleford who typed the original manuscript, an awesome task considering the many special symbols involved.

The Authors
May, 1964

CONTENTS

Burroughs Corporation

Second, revised edition

PREFACE

This book has grown out of the authors' efforts in teaching classes in both ALGOL 58 and Burroughs Extended ALGOL, an expanded version of ALGOL 60 implemented for the Burroughs B 5500 Computer. It also draws heavily upon their experience in actual programming for the B 5500 in Extended ALGOL. The book is aimed, primarily, at nonprofessional programmers. Therefore, the audience may very well include engineers, scientists, students, technicians, etc., who are interested in becoming proficient enough in ALGOL programming to write their own programs.

To attain its aim of providing the nonprofessional programmer with a working knowledge of ALGOL the text is based upon a proper subset of Burroughs Extended ALGOL. The subset is small enough to be readily grasped by the nonprofessional programmer. Yet it is comprehensive enough for most scientific and engineering applications. A knowledge of this subset may be amplified, at the discretion of the student, to include other features of Extended ALGOL.

A point of departure from previous textbooks on programming in Algorithmic languages is an emphasis on one of the most attractive aspects of ALGOL. This aspect is the precise definition of the language in terms of its syntax. It is consistency in this area that provides the continuity between the subset and Extended ALGOL. This text presents syntax gradually, as an integral part of the discussion of the language. The syntactical approach simplifies the learning of this subset and makes readable the reference material on ALGOL and Extended ALGOL.

The text begins by defining the basic elements of the language and subsequently develops the entire language, in a building block manner, from these elements. A simple form of input/output is introduced quite early, allowing the student to write complete, working programs as each new concept is developed. This is extremely effective in creating and maintaining interest. The liberal use of examples serves to point out the use of the constructs discussed. Each chapter is followed by a set of exercises which will allow the student to test his knowledge of the material covered, often by writing an actual program. These exercises are largely concerned with basic features of the language and can be supplemented by the instructor with applications-oriented problems.

The approach used in the text has been followed in several actual class situations and has proved highly successful. The availability of a computer for running student problems is an added advantage. Each chapter can be covered in a class of about three hours' duration. This would indicate about 30 hours of class time for adequate coverage of the material.

ALGOL

Programming

.....A Basic Approach

D. H. Thurnau, Ph. D. - Marathon Oil Company

R. E. Johnson - Burroughs Corporation

R. J. Ham - Burroughs Corporation

CHAPTER 1 - INTRODUCTION

1.1 THE COMPUTER

In the relatively short period of time since the advent of the digital computer, an almost revolutionary change has been wrought in the approach to problem solving and the processing of data.* This phenomenon is continuing to grow and now pervades, not only the physical sciences and obvious areas of data manipulation, but also the social and biological sciences and all levels of business decision making.

Digital computers cannot compete with their designers in the ability to solve complex problems. In fact, the machines themselves are not able to solve even the simplest problem without human help. The approach required to solve a problem expressed as a precise, step-by-step means of attaining the solution, must be provided to the computer. It is this set of step-by-step instructions that will be called a program. Programming a computer, then, is the development of such a set of instructions.

If the computer must be given instructions before it can effect the solution of a problem, where does its value lie?

Computers are not really very smart. The most complex tasks of which they are capable are the simple operations of arithmetic: addition, subtraction, multiplication, and division—but they do them so terribly fast! In fact, it is in its speed where much of the power of the digital computer is centered. For example, the B 5500 can perform as many as 200,000 additions in a single second. The computer is also capable of executing these arithmetic operations to a much greater degree of precision than would be feasible in any other practical means of computation. It is significant that

*The first electronic digital computer was built in the early 1940's with the first commercial machine available some ten years later.

the modern digital computer originated in the need to perform voluminous calculations, beyond the practical scope of existing equipment, in the development of the atomic bomb.

The digital computer does in fact possess one basic capability beyond that of the elementary arithmetic operations. It has the ability to make decisions on an either/or basis and take alternative courses of action. For example, the machine could be instructed to take one course of action if a variable X was negative and another if it was positive or zero.

One of the major differences between the early machines and present day digital computers lies in the concept of a stored program. Early machines depended upon some external source for their instructions. Modern computers are designed to store the sequence of instructions, as well as the input data and the results of computations, in their internal, rapid access, memory. This makes the computer much more versatile.

Other significant improvements lie in the rapid technological advances in the hardware itself and in the development of software. Software is defined as a group of one or more programs written to ease the task of the user in programming and operating the computer. The B 5500 ALGOL compiler is such a program.

In summary then, to utilize the power of the computer, a problem must be presented to the computer in terms of those basic operations which it is capable of performing. How this is done will be the subject of the following discussion.

This section closes with an amplification of the previous point regarding the basic abilities of the machine. A computer should be considered as a tool which extends the intellect of man. By freeing him from the drudgery of routine calculations and extending his ability to perform these calculations by many orders of magnitude, the computer allows man to utilize his creative ability to a greater extent. It is the merging of the capability of man and machine which opens the vistas of a limitless future.

B 5500 COMPUTER SYSTEM

1.2 PROBLEM SOLVING ON A COMPUTER

Since it has been seen that the computer must be provided with specific instructions as to how a problem can be solved, and since these instructions must be expressed in terms of the basic operations which the machine can perform, it will be beneficial to consider the subject of problem solving on a computer.

There are seven specific areas which should be considered when a digital computer solution to a problem is under development.

1. Problem definition and establishment of goals.
2. Mathematical description (model).
3. Reduction to a numerical process.
4. Programming.
    a. Flow charting
    b. Coding
5. Debugging.
6. Production.
7. Evaluation.

A critical factor in the ultimate success of any computer approach is the initial problem definition and the establishment of attainable goals. It is against these goals that success or failure will be measured. As Richard Hamming stated, "The objective of computing is insight, not numbers."* Appropriate problem definition allows the needed information to be gotten without also obtaining much extraneous information.

The desirability of defining a problem to get specific information does not preclude use of the computer as a research tool to produce results which cannot be anticipated. However, there should always be criteria for measuring the success of the effort.

Once the problem has been defined in broad terms, it must be expressed in quantitative terms. Often this quantitative form is called a mathematical model. An example of a mathematical model might be a single formula or it might be the description of a missile trajectory. The model is merely the description of a process in mathematical or precise numerical terminology.

After the problem is defined quantitatively, consideration must be given to whether or not the computer has the capability to perform all of the operations which are required. For example, does the problem require the extraction of a square root? Since the only operations which the computer can perform are the elementary arithmetic ones of addition, subtraction, multiplication, and division, the square root operation must be expressed in terms of these simple operations. This phase is called numerical analysis. Fortunately, most complex mathematical operations, as well as most methods of solving problems of a nonmathematical nature, can be reduced to simple numerical steps which the computer can handle with the basic operations and the capability to make comparisons.

*"Numerical Methods for Scientists and Engineers," Richard W. Hamming, McGraw-Hill.

The importance of basic knowledge in the area in which the problem is found cannot be overlooked. For example, it is not reasonable for someone who knows no Russian to attempt to devise a detailed technique for automatic translation of Russian to English. Similarly, in solving most scientific problems on a computer, there is no substitute for a sound background in mathematics.

With the problem solving process described as a step-by-step sequence of elementary operations, an equivalent series of instructions can be given to the computer and then executed to effect the solution of the given problem. It is this sequence of instructions which is the computer program.

There are two steps in the writing of the program. One is the expression of the numerical process in a formal manner, such as a flow chart. The second is the actual writing of the program in a language which the machine can accept. The first of these will be discussed briefly later in this chapter. The second is basically the subject of the entire text.

Once the program has been written, a period of time is usually required to make sure it is in fact performing correctly. This phase is usually called checkout or debugging. The programmer usually tries to eliminate as many errors as possible by carefully checking his program before it is ever given to the computer. The program is tested on the computer using input information or data for which the results are known.

When the program is checked out and running correctly, it can be used to calculate results based on realistic sets of data. This phase is considered as production and produces results which can be used to determine the success of the problem solving operation.

It is this evaluation which constitutes the final phase of the problem solving operation. Based on the original problem definition and the goals which were to be attained, production results can be used to measure the degree of success which was attained. If the objectives were reached the particular job is finished; if not, it may be necessary to reevaulate the original objectives and go through all or part of the problem solving cycle again.

Consider how the first three steps of the problem solving technique might be applied to a given problem. Suppose we define a problem and an objective by saying that we have two lines on a plane surface and want to know where they intersect.

Mathematically, the problem can be stated by describing each of the lines relative to an x-y coordinate system on the plane. The lines are then defined by linear equations. For example, line 1 would be represented by the equation $ax+by=c$ and line 2 by the equation $dx+ey=f$. The problem then becomes one of solving the two equations simultaneously. That is, values of x and y are needed which will satisfy both the equation for line 1 and the equation for line 2 at the same time.

However, we cannot merely provide these equations to the computer and await an answer. We must specify, in terms of the elementary operations, a procedure by which the computer can solve these equations.

It can be shown that the required values of x and y are given by the following expressions:

$$x = \frac{ec-bf}{ae-db}$$

$$y = \frac{af-dc}{ae-db} \qquad \text{where } ae-db \neq 0$$

Thus, the problem has been reduced to one of six multiplications, three subtractions, and two divisions, which can easily be implemented for solution on a computer.

1.3 ALGORITHMS

We have been discussing the idea of a step-by-step approach to a problem using only the elementary operations of arithmetic. Now we will describe this in a formal manner and give the concept a title—an ALGORITHM. There are many acceptable definitions of an algorithm, one of which is:*

> An algorithm may be defined as a series of simple operations which, when applied to a problem from a particular class of problems, will lead to a solution in a finite number of steps.

The close relationship between the idea of an algorithm as defined above and the development of a computer program should now be obvious. If an algorithm can be written for a problem which is being considered for a computer solution, the necessary computer program can be easily developed since the approach to the solution is spelled out in terms of simple operations by the algorithm.

Consider again the problem of solving two linear equations simultaneously. It was stated that the values of x and y could be found from the relationships:

$$x = \frac{ec-bf}{ae-db} \qquad y = \frac{af-dc}{ae-db}$$

These equations provide the basic approach to finding x and y.

To develop more fully the idea of an algorithm, that is the precise, step-by-step approach to finding x and y, the specific steps to follow might be written:

*"Algorithms and Automatic Computing Machines," B. A. Trakhtenbrot, D. C. Heath & Company.

1. Calculate the value of ae and save it.
2. Calculate the value of db and save it.
3. Subtract the result of step 2. from the result of step 1. and save it.
4. Calculate the value of ec and save it.
5. Calculate the value of bf and save it.
6. Subtract the result of step 5. from the result of step 4. and save it.
7. Calculate the value of af and save it.
8. Calculate the value of dc and save it.
9. Subtract the result of step 8. from the result of step 7. and save it.
10. Divide the result of step 6. by the result of step 3. and call the resulting value x.
11. Divide the result of step 9. by the result of step 3. and call the resulting value y.

These 11 steps represent the algorithm for finding x and y. A person totally unfamiliar with the problem and knowing how to subtract, multiply and divide could find x and y, given the above step-by-step approach. The task of a programmer then, is to put the steps of an algorithm into a form which is meaningful to the computer. Since the medium in which the algorithm is expressed is unimportant, it is obvious that a computer program is in fact an algorithm, since it expresses a specific, step-by-step approach to solving a particular kind of problem.

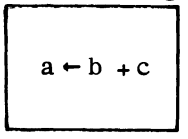
1.4 FLOW CHARTING

Once the mathematics of a given problem have been stated in terms of an algorithm, the task of expressing the algorithm in terminology which the computer understands can begin. The result of this expression is a computer program. However, good programming technique often calls for a step preceding that of actual program writing. This step is called flow charting.

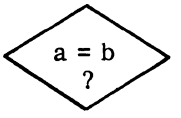
A flow chart is a block diagram indicating the logical flow of a program. That is, it presents in a pictorial form the sequence of steps leading to the solution. It is based to a large degree upon the algorithm or numerical process which has previously been developed, representing it in a highly formal manner. It will usually, however, be more comprehensive than the algorithm, including all operations which supply valid data to the algorithm and results to the programmer.

The technique used in flow charting tends to vary with the individual; however, the following conventions might be suggested:

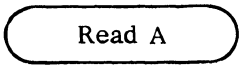
An overt action such as the addition of two numbers and the sav: of the result is represented by a rectangle.



A test or comparison is represented by a diamond.

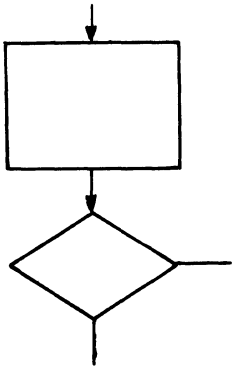


Input/Output operation denoted by an elipse.



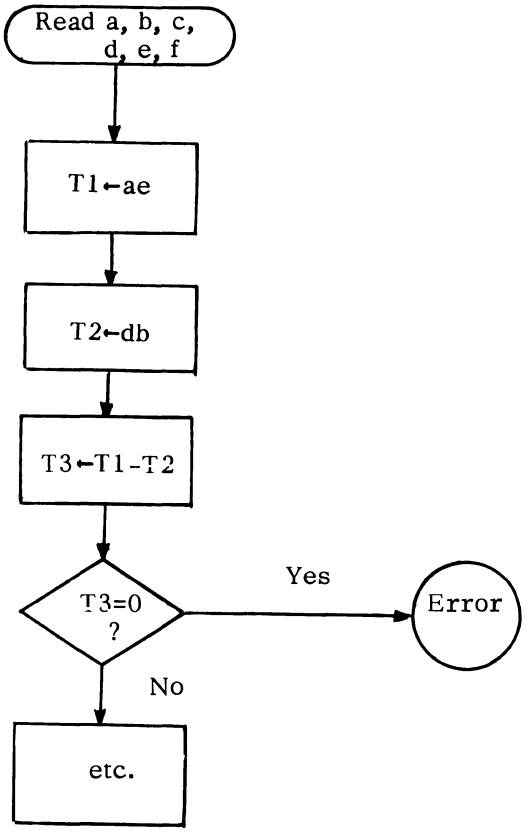
Connector to a distant point in the flow chart indicated by a circle.

The flow through these is indicated by lines from box to box with arrow heads denoting directions.



Note the single line going into the decision box and the two lines coming from it. This box usually produces either a yes or a no answer and shows both alternatives.

The previous algorithm might be flow charted as follows:



1.5 COMMUNICATING WITH THE COMPUTER

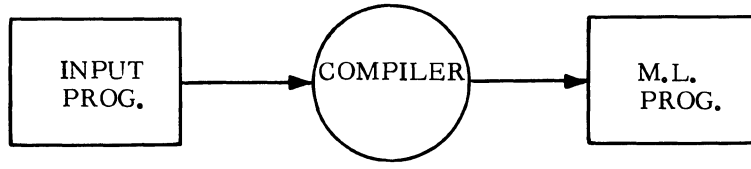
In one sense, the computer program was created when the flow chart was developed. It fits most of the criteria since it is broken down into simple operations in a proper sequence leading to a solution. However, the computer is not capable of understanding the program in this form. Therefore, the flow chart must be transcribed into a form which the computer will accept.

Every computer has its own internal language. Usually this language, while highly appropriate and meaningful to the given machine, is highly foreign to a human programmer. In fact, it is called machine language.

To make communication easier for a human problem solver, languages have been developed which can be mechanically converted into machine language. These are called procedure oriented languages since they are ways in which well defined procedures (i.e.,

flow charts, etc.) can be expressed in a form palatable to the computer.

The machine itself effects the conversion of the program to its internal language through a program called a compiler. The compiler is itself a program written for the particular machine, usually by the manufacturer. A particular program which is to be translated (or compiled) is provided to the compiler as data. This result (or output) of the compilation is a machine language equivalent of the input program. This operation might be depicted as follows:



The input program is usually called the source program, and the resulting machine language is called the object program.

This text discusses a particular procedure oriented language which is a subset of Burroughs Extended ALGOL.* This in turn is derived from the ALGOL 60 language. Actually the language presented (except for Input and Output, which are not defined in ALGOL 60) is essentially a subset of ALGOL 60. Thus, it is useful to all students of the ALGOL language.

The ALGOL language had its beginnings in an international desire to develop a general, precisely defined, algebraic language for use in communicating problems to a computer. In 1958, a meeting in Europe produced a first attempt at such a language called ALGOL 58. ALGOL, by the way, is an acronym developed from the two words ALGOrithmic Language. This acronym calls attention to the important connection between algorithms and programming languages.

In 1960 the group met for a second time to improve upon the language and to further solidify its logical foundations. The resulting language is called ALGOL 60.**

The outstanding features of the language developed by this group are its consistency and the precise definition of its syntax. The syntax is the body of rules which governs the construction of meaningful entities from the basic language elements, much as the syntax of English tells us how to form sentences, paragraphs, etc. from words.

ALGOL bears a strong resemblance to algebraic notation. This, of course, makes it highly adaptable to use in scientific and engineering problems. For example, the simple algebraic expression for calculating distance in terms of rate and time,

*"Extended ALGOL Reference Manual for the Burroughs B 5000," 5000-21012, Burroughs Corporation.

**"Report on the Algorithmic Language ALGOL 60," Naur, P. et al., Communications of the Association for Computing Machinery, May, 1960.

$$d = rt$$

where d stands for distance, r for rate and t for time, would appear in ALGOL as

$$d \leftarrow r \times t$$

where d represents the variable distance, $\leftarrow$ stands for the operation of assignment, r and t are the variables rate and time and $\times$ says to multiply.

That is, the ALGOL "statement" says to take the value of r, multiply it by the value of t and assign it to the variable represented by d.

A program written in ALGOL is easily read and readily changed. Since the language is so much like algebraic notation, the initial writing of the program is itself quite simple and the basic mathematics of his problem are constantly before the user. Further, programming in ALGOL requires very little intimate knowledge of the machine that processes the program. Quality results are thus attainable by the engineer or scientist without his becoming immersed in details that are totally foreign to his problem.

1.6 SYNTAX

To facilitate the discussion and the formal definition of the ALGOL language, a simple symbolic language was developed. This symbolism provides a means of discussing a programming language, just as mathematics provides a notation for dealing with quantitative problems. This precise definition of the language is extremely important in implementing a compiler to translate the source language into machine language, since the syntax expresses the rules which are to be embedded in the compiler. If a compiler is written for a poorly defined language, the true syntax of the language accepted by the compiler is available only to those who can read the compiler program. The syntax of a programming language is similar in function to the syntax of the English language, in that it specifies those rules which govern the writing of meaningful constructs in the language.

The symbolic language which is used is called a metalanguage—that is, a language which is used to define another language. It is based upon a very few symbols which are defined as follows:

1. The Broken Brackets, $\langle\ \rangle$.

   An element of the language which is not a basic symbol of the language will be called a metalinguistic variable and will appear between broken brackets.

   Example, $\langle$number$\rangle$

In other words, when a number is defined or used in the definition of some other construct, it will appear as ⟨number⟩.

2. The Colon-Colon-Equal Symbol, ::=.
   This symbol is read "is defined as" and is used to separate a metalinguistic variable on the left from the expression appearing on the right which defines this variable.
   Example, ⟨number⟩ ::=
   This would be read, "a number is defined as."

Often there are alternative definitions for a given variable. This will require an additional symbol.

3. The Vertical Line, |.
   This symbol is read "or" and is used to separate alternative definitions of a metalinguistic variable. For example, consider how the concept of color might be defined in this manner.
   Example, ⟨color⟩ ::= ⟨bright⟩ | ⟨dull⟩
   This would be read, "color is defined as bright or dull."

When a definition has been reduced to the basic elements of the language (i.e., the alphabet or a similar basic element), those basic elements will not appear enclosed in broken brackets. This is, in fact, how the basic elements are distinguished. These elements will not always be in the nature of single characters but may be groups of characters which are considered as a basic entity in themselves. For example, if red, yellow, and orange are basic elements in a language, the following might be written:

⟨bright⟩::= red|yellow|orange

One further set of symbols is required to completely define ALGOL in this manner.

4. The Braces, { }.
   The braces are used to define variables which cannot be expressed in terms of previously defined variables or groups of variables. The English language definition will appear between the braces.

This method of definition is not restricted in its application to the discussion of programming languages. To develop this syntactical approach further, consider how a telephone number might be rigorously defined.* To start with, the basic symbols used in the definition of a telephone number must be given.

⟨basic symbol⟩ ::= ⟨letter⟩ | ⟨digit⟩ | ⟨hyphen⟩

This is read, "a basic symbol is defined as a letter or a digit

*Courtesy Glynn Jones, Burroughs Corporation.

or a hyphen." Since the definition itself contains metalinguistic variables, it is necessary to present further definitions:

⟨letter⟩ ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|R|S|T|U|V|W|X|Y

⟨digit⟩ ::= 0|1|2|3|4|5|6|7|8|9

⟨hyphen⟩ ::= -

Since the elements on the right side of the above metalinguistic formulas are not enclosed in broken brackets, it can be correctly assumed that they represent the basic elements from which a telephone number is written. Notice that the definition of letter excludes the letters Q and Z. This is perfectly valid since the definition specifically shows this and because these letters do not enter into a telephone number.

Once the basic elements have been specified, it is an easy task to define a telephone number in a syntactical manner.

⟨telephone number⟩ ::= ⟨area code⟩⟨exchange part⟩⟨hyphen⟩⟨subscriber number⟩

A telephone number is defined as an area code followed by an exchange part, followed by the basic symbol hyphen and followed by the subscriber number. For readability, spaces are specifically allowed between the above components. Notice that the juxtaposition of metalinguistic variables in the definition serves to indicate that these variables appear side by side in the element defined.

⟨area code⟩ ::= ⟨digit⟩⟨digit⟩⟨digit⟩|⟨empty⟩

⟨empty⟩ ::= {the null string of symbols}

The area code may be composed of three consecutive digits or it may be empty, indicating that its use is not mandatory.

⟨exchange part⟩ ::= ⟨exchange⟩⟨frame number⟩

⟨exchange⟩ ::= ⟨letter⟩⟨letter⟩|⟨digit⟩⟨digit⟩

Notice that the exchange may be made up of two consecutive letters or two consecutive digits.

⟨frame number⟩ ::= ⟨digit⟩

⟨subscriber number⟩ ::= ⟨digit⟩⟨digit⟩⟨digit⟩⟨digit⟩

The above syntactical definition serves to validate the structure of any telephone number. Consider for example, the telephone number 303 623-3284. The area code which may consist of three digits is 303. The exchange part is defined as the exchange followed by the frame number. Since the exchange may be made up of two digits and the frame number a single digit, 623 is structurally correct for the exchange part. Following the hyphen, which is a basic symbol, is the subscriber number 3284 which satisfies the definition. Verify that the following telephone numbers satisfy the syntactical definition.

CH4 - 6200
313 OL5 - 1112
789 - 1234
113 221 - 4605
412 PR4 - 8241
MA3 - 3284

Verify that the following are not correctly formed telephone numbers:

COL - 1643
794 , 2601
GA 2-21 21
ZE5 - 7163

The purposes of learning the syntactical definition of ALGOL are many. It is an extremely useful tool in merely learning the language discussed in this text. Furthermore, a knowledge of syntax will allow the programmer to extend his knowledge of ALGOL, by referring directly to reference manuals on the language, the majority of which are written using this notation. Finally, it facilitates the debugging or error correction of a program since all possible constructs may be checked, using the appropriate syntactical defnition. It would be impossible to define by example all constructs allowed in ALGOL.

EXERCISES

1. Write an algorithm for finding, from a group of five numbers, the number with the smallest absolute value

2. Write an algorithm for algebraically sorting five numbers into ascending order.

3. List the steps involved in solving the quadratic equation, using the quadratic formula:

$$X = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$$

4. Develop a flow chart for the problem expressed in exercise 1 above.

5. Construct a flow chart for the problem given in exercise 2 above.

6. Flow chart the algorithm to solve a quadratic equation.

7. Develop syntactical formulas for writing a date in forms, such as:

   Nov. 18, 1957
   28 Sept. 1956
   22 Dec 24
   June 5, 1904
   26 January 1899
   Sat., Jun 13, 1964

CHAPTER 2 - BASIC LANGUAGE ELEMENTS

2.1 CHARACTER SET

ALGOL, like any language, is written in terms of letters, numerals and special symbols. The number of these characters that can be used with present-day computing equipment is limited by mechanical practicalities. One result of this is that lower-case letters are rarely used. Another is that the number of available special symbols is not large.

The B 5500 utilizes a set of 64 characters, which includes:

1. The 26 letters of the English alphabet, in upper case,
2. The 10 Arabic numerals (digits),
3. The blank (a single space), and
4. The following special symbols:

[. (; :) ,] ← + - × / < ≤ = ≠ ≥ > $ # @ % * & " ?

There are two important concepts here that are unusual to the layman. One is the status of the blank as a character. While one or more blanks are often used in programs merely to improve readability, there are a few places where blanks are essential to meaning. This will be discussed further in Section 2.3. The other concept is the strict limitation to just this set of characters, with no variations such as superscripts or subscripts.

Two of the special symbols, given above, are used for very restricted purposes. The quote mark is used only in conjunction with strings, which are covered in Section 7.2. The question mark may not be used at all in writing ALGOL programs. It is normally used only by the B 5500 system to represent an illegal hole combination on a punched card.

2.2 BASIC SYMBOLS

It is essential to classify the basic symbols from which ALGOL programs are formed. Without this, precision, clarity and completeness are not possible in discussing and defining many aspects of the language. There are three kinds of basic symbols. These are letters, digits and delimiters. Expressed as a formula, this becomes:

⟨basic symbol⟩ ::= ⟨letter⟩|⟨digit⟩|⟨delimiter⟩

The first two forms are defined as follows:

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

⟨digit⟩ ::= 0|1|2|3|4|5|6|7|8|9

It is helpful that these definitions correspond, for the most part, with those normally associated with "letter" and "digit." Note that many other "letters," such as lower-case letters, the Greek alphabet, etc., are not included as letters in B 5500 ALGOL.

It is necessary to discuss the third form of basic symbol, the delimiter, somewhat formally. The role of the delimiters in the language is classified in the remainder of this section. The specific meaning of each one will be presented in subsequent sections of the text. A full appreciation for this classification depends upon an overall acquaintance with ALGOL.

The special symbols of the B 5500 character set are used as delimiters, but there are many delimiters that cannot be represented by special symbols. In practice this requires the adoption of a concept known as "reserved words." The latter are groupings of letters, such as STEP, GO, FOR, OUT, IF, which have fixed meaning in the language. It is useful to think of each of them as an additional special symbol.

However, even if each of them could be represented by a special symbol, such would not be desirable because their presence lends to ALGOL much of the readability of English.

If reserved words could be written in italics or bold-face type, they would not have to be reserved. Since letters come in only one style, it is essential to avoid the use of reserved words for any purpose not defined in the language. A complete list of the reserved words of Extended ALGOL is given in Appendix A.

A broad definition of delimiters might assert that they constitute the permanent vocabulary of ALGOL. That is, they specify definite processes or meanings that are to be applied to adjacent entities. For example, the special symbol × specifies that the quantities given on either side of it are to be multiplied together. The parentheses are used in several ways to cause a group of entities to be treated as one entity, as in 2 ×(A × B +1). Similarly, the delimiters BEGIN and END (reserved words) are used to enclose several constructs so that they can be treated as a single entity.

Spelling out the exact meaning of each of the ALGOL delimiters is essentially the purpose of this text. For the present it is

sufficient to complete our defininition of basic symbol with only cursory explanations of the delimiter functions.

⟨delimiter⟩ ::= ⟨arithmetic operator⟩ | ⟨relational operator⟩ | ⟨logical operator⟩ | ⟨sequential operator⟩ | ⟨separator⟩ | ⟨bracket⟩ | ⟨declarator⟩

This merely says there are seven kinds of delimiters. Happily, some of these are defined so as to coincide with normal technical usage.

⟨arithmetic operator⟩ ::= + | - | × | / | *

These are used in the manner of algebraic notation with the addition of the * operator to denote exponentiation.

⟨relational operator⟩ ::= ≤ | < | = | ≥ | > | ≠

These are used to assert relationships between arithmetic quantities. These assertions constitute truth values that are needed in making yes/no decision.

⟨logical operator⟩ ::= OR | AND

These are used in forming combinations of truth values, the final truth or falsity of which is usable for decision making.

⟨sequential operator⟩ ::= GO | TO | IF | THEN | ELSE | FOR | DO | READ | WRITE | PAGE

These specify courses of action to be taken when used in language constructs that are unfamiliar at this point.

⟨separator⟩ ::= , | . | @ | : | ; | ← | ⟨single space⟩ | STEP | UNTIL | COMMENT

⟨single space⟩ ::= {a single unit of blank horizontal spacing}

These play a role in ALGOL similar to that of punctuation and spacing in English.

⟨bracket⟩ ::= [| (|) |] | " | BEGIN | END

As stated above, these are used to enclose combinations of entities to give them special meaning or treatment.

⟨declarator⟩ ::= INTEGER | REAL | ARRAY | LABEL | LIST | FORMAT | IN | OUT | FILE | MONITOR | DUMP | PROCEDURE

These delimiters essentially inform the reader (or the compiler) that certain entities are hereby defined and/or treated as having certain specific properties.

Much more detail will be given on all of these delimiters as their uses are encountered throughout the text.

2.3 IDENTIFIERS

A powerful feature of ALGOL is the possibility of using problem-oriented nomenclature in writing computer programs. This permits programs to be largely self-documenting and highly readable. With little restriction, names may be devised to represent many different entities. For instance, variables are normally given names which, in themselves, describe the arithmetic quantities associated with them. All such names are formally called identifiers.

The exact usage of identifiers in various ALGOL constructs will be discussed in detail as these constructs are encountered. However, in every case, the metalinguistic variable identifier will be used to mean the same thing. The rules for forming an identifier are expressed by the formula,

⟨identifier⟩ ::= ⟨letter⟩ | ⟨identifier⟩⟨letter⟩ | ⟨identifier⟩⟨digit⟩

This specifies that an identifier is composed of any combination of letters and digits, such that the leftmost character is a letter.

Practical considerations in the B 5500 computer impose a minor restriction that the total number of characters in an identifier may not exceed 63. One further restriction is that reserved words (see Appendix A) cannot be freely used as identifiers.

Note that the above definition specifies only letters and digits as legal characters in an identifier. Thus, blanks and special symbols may not occur in identifiers. This makes it possible to distinguish identifiers from other ALGOL constructs. For example, in A + B2-CG, the three identifiers A, B2 and CG are readily recognized. Similarly, A STEP 2 UNTIL ZIT is easily found to contain two identifiers, A and ZIT, two reserved words, STEP and UNTIL, and one unsigned integer, 2 (to be defined in Section 2.4).

In the latter example, note that removal of the blanks (used as separators) produces an entity, ASTEP2UNTILZIT, which can only be regarded as a single identifier. This is a direct consequence of the fact that reserved words are formed from the same kind of characters that are used in identifiers. In fact, the only mandatory occurrences of blanks in B 5500 Extended ALGOL are before reserved words which follow other reserved words and identifiers, and after reserved words which precede either identifiers or unsigned numbers. Three of these requirements are illustrated in A STEP 2UNTIL ZIT, where 2UNTIL will not be mistaken for an identifier, whereas ASTEP, STEP2 or UNTILZIT would lose their intended meanings.

The reader may verify the classification of the examples given in the following lists:

Legal Identifiers	Illegal Identifiers
A	BEGIN
I	1776
B5	2BAD
YSQUARE	$
TOOBAD	X/Z
LONGTONS	Y SQ
LAZY8	W-2FORM
PRESSURE	∞
XOVERZ	⟨CAPTION⟩
D2P471GL	SEC(X)
CURRENT	RATE/HR
SPEED	NO.
ALTITUDE	

2.4 NUMBERS

Numerical values which are built into an ALGOL program are called numbers. They are sometimes called constants because they are not altered during execution of a program.

A number in ALGOL may not be written in all of the forms of number in common technical usage. Further, there are character set limitations which interfere with normal ways of writing powers of 10. The meaning of number must be carefully spelled out.

The rules for correctly forming a number are completely given in the following series of syntactical definitions:

⟨number⟩ ::= ⟨unsigned number⟩ | + ⟨unsigned number⟩ | -⟨unsigned number⟩

⟨unsigned number⟩ ::= ⟨decimal number⟩ | ⟨exponent part⟩ | ⟨decimal number⟩⟨exponent part⟩ |

⟨decimal number⟩ ::= ⟨unsigned integer⟩ | ⟨decimal fraction⟩ | ⟨unsigned integer⟩⟨decimal fraction⟩

⟨exponent part⟩ ::= @⟨integer⟩

⟨decimal fraction⟩ ::= .⟨unsigned integer⟩

⟨integer⟩ ::= ⟨unsigned integer⟩ | +⟨unsigned integer⟩ | -⟨unsigned integer⟩

⟨unsigned integer⟩ ::= ⟨digit⟩ | ⟨unsigned integer⟩⟨digit⟩

The first formula states that a number may or may not carry a sign. An unsigned number is understood to be positive. Later formulas indicate that a number is a base 10 number, with or without an exponent part. It may also consist of the exponent part alone, indicating that the number is a power of ten. (Thus, 1.0@6, 1@6, +10@5, @6 all are possible ways of writing 1000000.)

The reader will profit from a verification of the following examples of each of the meta-variables defined above. A series of illegal numbers is also given.

Unsigned integers	Decimal fractions	Decimal numbers
5	.5	.69
69	.69	.546
	.013	3.98

Integers	Exponent parts	Unsigned numbers
1776	@8	99.44
-62256	@-06	@-11
548	@+54	1354.543@48
		.1964@4

Numbers	Illegal numbers
0	5000.
+549755813887	1,505,278.00
1.75@-46	@ 63
-4.314@68	5 @8
-@2	1@2.5
.375	1.667E-01

The illegal number examples, given above, emphasize the fact that the only characters that are used to form numbers are digits and the basic symbols . @ + and - . Note that no provision is made for spaces to occur inside numbers. Also, note that a decimal point as the final character of a number is not allowed.

For purposes which will later become clear, numbers are classified into two types, INTEGER and REAL. All integers are of type INTEGER. All other numbers are of type REAL.

The B 5500 hardware allows a maximum absolute value for integers of 549755813887. Non-zero REAL numbers must lie in the approximate (absolute value) range 1.75@-46 to 4.314@68.

2.5 VARIABLES

A variable in ALGOL has a role similar to that of a variable in algebra. It is a symbolic representation of an arithmetic quantity. A variable may be thought of as the name (temporarily) attached to a specific storage location. The arithmetic value stored there may be altered by the program, and it may be used, without destruction or erasure, for a variety of computational purposes.

The syntactical definition

⟨variable⟩ ::= ⟨simple variable⟩ | ⟨subscripted variable⟩

indicates that there are two basic kinds of variables. However, the definition and discussion of subscripted variables will be postponed until Chapter 6.

The rule for forming a simple variable is given by:

⟨simple variable⟩ ::= ⟨identifier⟩

Note that this does not say that all identifiers are simple variables. Rather, it means that simple variables are formed by the rules already given in Section 2.3 for identifiers.

The information that a given identifier is a simple variable in a given program is provided by a type declaration which precedes any use of this identifier in the program. (This and other declarations are discussed further in Section 3.2.) The type declaration also specifies whether the kinds of arithmetic values represented by a simple variable are inherently integers or not. For reasons of precision and efficiency, it is usually desirable, in digital computing, to specify those variables whose values must always be integers. (If the integral value could possibly become greater than 549755812887, this practice cannot be followed.) Such variables are declared to be of type INTEGER. Variables whose values are not necessarily integers are declared to be of type REAL.

The rules for writing a type declaration are given by:

⟨type declaration⟩ ::= ⟨type⟩⟨type list⟩
⟨type⟩ ::= REAL | INTEGER
⟨type list⟩ ::= ⟨simple variable⟩ | ⟨type list⟩,⟨simple variable⟩

Thus, type declarations are permitted to be written either as

```
INTEGER     I8
REAL        W
REAL        A
INTEGER     K
REAL        MAX
```

or as

```
INTEGER     I8,K
REAL        A, W, MAX
```

Either group of declarations means the same thing. Namely, I8 and K are to be treated as representing integral arithmetic values, and A, W and MAX are understood to represent values which (may, or) may not be integers.

The arithmetic values that can be taken on by variables are subject to the same limitations given for numbers in Section 2.4.

2.6 COMMON FUNCTIONS

Because of their frequent occurrence in scientific programming, a number of common mathematical functions are furnished automatically by the ALGOL compiler. These are listed in the table below. The last three listed are not always regarded as mathematical functions but are best represented as such in an algorithmic language. The nontechnical student may wish to ignore all but the last three common functions.

These functions are furnished in the form of subprograms that operate on the argument to produce the desired result. The square root is computed by an iterative process. The trigometric, logarithmic and exponential functions are computed from polynomial approximations.

The argument of each of these functions may be an arithmetic expression, whose formal definition is given in Section 2.7. For brevity, the symbol AE will stand for arithmetic expression in the table to follow.

COMMON FUNCTIONS

ALGOL Representation	Description
SQRT (AE)	Produces the square root of the value of AE
SIN (AE)	Produces the sine of the value of AE
COS (AE)	Produces the cosine of the value of AE
ARCTAN (AE)	Produces the principal value of the arctangent of the value of AE
LN (AE)	Produces the natural logarithm of the value of AE
EXP (AE)	Produces the exponential function of the value of AE, i.e., e^{AE}.
ABS (AE)	Produces the absolute value of AE
SIGN (AE)	Produces one of three values depending on the value of AE (+1 for AE>0, 0 for AE=0 and -1 for AE<0)
ENTIER (AE)	Produces the value which is the largest integer not greater than the value of AE

The ENTIER function is easily misunderstood to perform simple truncation. The following examples show that this is not the case.

ENTIER (2.6) = 2; ENTIER (3.1) = 3;
ENTIER (-0.01) = -1; ENTIER (-3.4) = -4;
ENTIER (-1.8) = -2.

Since the common (base 10) logarithm is not one of the functions built into the ALGOL compiler, it is useful to be aware of the relationship LOG_{10} (X) = LOG_{10} (e) × LN (X)
where LOG_{10} (e) has an approximate value of 0.4342944819.

There is a type associated with all of the above functions except ABS, which has the type of its argument. The rest of the functions are considered to be of type REAL except SIGN and ENTIER which are of type INTEGER.

For SIN, COS and ARCTAN, the associated angles are in units of radians. Thus the ARCTAN function can have values from $+\pi/2$ to $-\pi/2$. Recall that the number of degrees per radian is $180/\pi$ or, roughly, 57.295780.

2.7 ARITHMETIC EXPRESSIONS

At this point, enough basic language elements have been defined to permit introduction of the primary active element of the language. This is the arithmetic expression. An arithmetic expression provides the rule for computing a single numerical result. Basically, it is composed of arithmetic primaries and operators.

The value of an arithmetic expression is obtained by performing the indicated arithmetic operations upon the actual numerical values of the primaries. The evaluation of an arithmetic expression is closely analogous to the evaluation of algebraic expressions. The exact rules by which the computer performs such an evaluation are spelled out later in this section.

A primary is defined as follows:

⟨primary⟩ ::= ⟨unsigned number⟩ | ⟨variable⟩ |
⟨function designator⟩ | (⟨arithmetic expression⟩)

Here we find that two of the possible forms for a primary involve metalinguistic variables not yet defined. One is an arithmetic expression enclosed in parentheses. Another is the function designator which, for the present, may be regarded merely as the formal

name given to the common functions of Section 2.6. Later, in Chapter 9, an expanded definition of function designator will be given.

⟨function designator⟩ ::= ⟨common function identifier⟩
(⟨arithmetic expression⟩)

⟨common function identifier⟩ ::= SQRT | SIN | COS | ARCTAN | LN |
EXP | ABS | SIGN | ENTIER

Note that the argument of the common function identifier is an arithmetic expression enclosed in parentheses.

We now see that we must know how an arithmetic expression is defined before we can write a primary which is other than an unsigned number or a variable. Yet, as will be seen, an arithmetic expression cannot be written without using a primary. Evidently the simplest arithmetic expressions must involve primaries which are either variables or unsigned numbers. Indeed, we shall see that the following are legal, if simple, forms of arithmetic expressions.

MEAN	LN (14.92)
22	(X)
SIN (ALFA)	NETMANHOURSWORKEDPERYEAR

Two kinds of arithmetic operators are explicitly defined.

⟨adding operator⟩ ::= + | -
⟨multiplying operator⟩ ::= × | /

These have the normal mathematical meanings: addition, subtraction, multiplication and division. Another mathematical operator, *, used for exponentiation, is defined in a table below.

We are now adequately prepared for the definition of an arithmetic expression. For completeness, the definition of a primary is repeated here.

⟨arithmetic expression⟩ ::= ⟨term⟩ | ⟨adding operator⟩⟨term⟩ |
⟨arithmetic expression⟩
⟨adding operator⟩⟨term⟩

⟨term⟩ ::= ⟨factor⟩ | ⟨term⟩⟨multiplying operator⟩⟨factor⟩

⟨factor⟩ ::= ⟨primary⟩ | ⟨factor⟩*⟨primary⟩

⟨primary⟩ ::= ⟨unsigned number⟩ | ⟨variable⟩ | ⟨function designator⟩ |
(⟨arithmetic expression⟩)

All of the rules for forming legitimate arithmetic expressions are concisely given here. Since arithmetic expressions are written so nearly like algebraic notation, the chief use the programmer might make of the above rules is to avoid writing illegal arithmetic expressions. For example, implied multiplication is not allowed,

nor can any other symbol, but ×, be used to indicate multiplication. Thus, A(B+C) and A.(B+C) are illegal ways of writing A×(B+C). Likewise, the syntax tells us that two arithmetic operators may not occur in sequence, such as W×-4 (which can be written legally as W×(-4) or -W×4 or -4×W).

The correct classification of examples in the following lists should be verified by the student.

Legal Examples	Illegal Examples
Primaries:	
5.678	+7
SIGMA	SIN X
Y1	A/B
(X-Y)	LOG((M-1)/10)
COS(T)	-Z
ABS(1-X/Y)	+(X-Y)
((GEE+HAW)/PLOU)	
(A×64*2+B)	
Factors:	
5.678	-9.81
CHARLIE	+DC8
2*(X+Y)	B-A
Y*3	X*-3
Q*V*2	10-16
(14+3.142)	
Terms:	
5.678	-13.6
MABLE	-(A+B)
K×F2	A+B
SUM/N	L*-A
4×A×(1/C)	*ENTIER (60)
(A+B)/(C-D)	RATE . DAYS
2*(X+Y)	4(AC)
Arithmetic Expressions:	
COS (A+B) +C	
Y*3	
+8	
B*2-4×A×C	
(-B+SQRT(D))/(A+A)	
-T*3	
5.678	
THETA	

There will be instances where the type of an arithmetic expression affects the results obtained in particular ALGOL constructs. The type may be ascertained by (perhaps repeated) use of the following table:

Operand On Left	Operand On Right	Value Resulting from the Operator		
		+,-,×	/	*
Integer	Integer	Integer	Real	Note 1
Integer	Real	Real	Real	Real
Real	Integer	Real	Real	Real
Real	Real	Real	Real	Real

Note 1: If the right operand is negative, Real, otherwise Integer.

The exact operation corresponding to the exponentiation operator, *, depends upon the types and values of the operands involved. These evaluation methods are specified for the factor Y*Z in the following table:

	If Z is Integer and			If Z is Real and		
	Z>0	Z=0	Z<0	Z>0	Z=0	Z<0
If Y>0	Meth.1	1	Meth.2	Meth.3	1	Meth.3
If Y<0	Meth.1	1	Meth.2	Note 1	1	Note 1
If Y=0	0	Note 1	Note 1	0	Note 1	Note 1

Method 1: Y*Z = Y×Y×Y ... ×Y(Z times)
Method 2: Y*Z = the reciprocal of Y×Y ... ×Y (Z times)
Method 3: Y*Z = EXP (Z×LN(Y))
Note 1: Value of expression is not defined

Present-day means of communicating with computers (e.g., punched cards) require the linearized mathematical notation represented by ALGOL arithmetic expressions. However, this causes ambiguities in expressions such as A+B/C or X*2+B since each has two possible mathematical interpretations:

$A + \frac{B}{C}$ or $\frac{A+B}{C}$ and X^{2+B} or $X^{2}+B$

To circumvent these ambiguities, it is necessary to assign a precedence to the arithmetic operators. Each operator has an order of precedence associated with it as follows:

First: *
Second: ×,/
Third: +,-

When operators have the same order of precedence, the sequence of operation is determined by the order of their appearance, from left to right. Parentheses can be used in normal mathematical fashion to override the usual order of precedence.

To help illustrate the importance of operator precedence as well as to provide some final concrete examples, the following table is offered:

Mathematical Expression	Equivalent ALGOL Expression	Nonequivalent ALGOL Expression
$A \times B$	A × B	AB
$A + \frac{B}{2}$	A + B/2	
$\frac{X + 1}{Y}$	(X + 1)/Y	X + 1/Y
$\frac{D + E^2}{2A}$	(D + E * 2)/(2 × A)	(D + E * 2)/2 A
$4(X + Y)^3$	4 × (X + Y)*3	4 × X + Y * 3
$\frac{M - N}{(M + N)^P} + 5 \times 10^{-6}$	(M - N)/(M + N)*P + 5@-6	

EXERCISES

Write ALGOL arithmetic expressions corresponding to each of the following mathematical expressions:

1. $x + y^3$

2. $(x + y)^3$

3. $x^{1.667}$

4. $A + \frac{B}{C} - 1570.795 \cdot 10^{-3}$

5. $\frac{A + B}{C}$

6. $A + \frac{B}{C + D}$

7. $\frac{A + B}{C + D}$

8. $\frac{A + B}{C + D} + X^2$

9. $\cfrac{A + B}{C + \cfrac{D}{F + G}}$

10. $1 + x + \frac{x^2}{2!} + \frac{x^3}{3!}$

11. $\left(\frac{P_1}{P_2}\right)^{q-1}$

12. $a \cdot b \cdot c^d - \left(2^x\right)^{-2}$

13. $(x_1^3 + x_2^3 + x_3^3)$

14. $2 \cdot P \cdot R \cdot \sin(\pi / P)$

15. $2R \sin \frac{A}{2}$

16. $2\sqrt{y^2 + (4x^2/3)}$

17. $\frac{-\cos^4 x}{4}$

18. $$\cfrac{x}{1 + \cfrac{x^2}{3 + \cfrac{(2x)^2}{5 + \cfrac{(3x)^2}{7 + (4x)^2}}}}$$

19. $\frac{A}{B \cdot C} + \frac{1964.}{\frac{B}{C}}$

20. (start time) + $\frac{\text{(distance traveled)}}{\text{(average speed)}}$

CHAPTER 3 - STRUCTURE OF ALGOL PROGRAMS

3.1 A PROGRAM

In Chapter 2 the basic elements of the ALGOL language were developed. The purpose of this chapter is to utilize these basic elements as building blocks in the development of the main objective, a series of instructions to the computer which constitutes a program. Recall from Chapter 1 that the actual computer program is a representation of the steps required to solve a particular problem in a highly formal manner. Once the program has been stored in the memory of the computer its execution is entirely dependent upon the internally stored instructions.

Just as it was necessary to be very precise in defining the basic language elements in the previous chapter it will be necessary to formalize the definition of a program beyond that which we have been using. In fact, all definitions will be developed in the same formal manner.

For the purpose of discussing the structure of a program we will introduce a symbolic notation for the elements that make up the program. These will not be the basic elements which have already been considered but rather constructs made up of these basic elements. A construct called a declaration will be noted by a D and one called a statement by an S. Both of these constructs will be discussed in full in Section 3.2.

A program might be defined as a bracket symbol BEGIN followed by declarations and then statements with the end of the program indicated by the bracket symbol END and a period which has been defined as a separator. Symbolically a more concise definition can be written as follows:

BEGIN D; D; S; S; S END.

A more general form would be:

BEGIN D; D; ... D; S; S; ... S; S END.

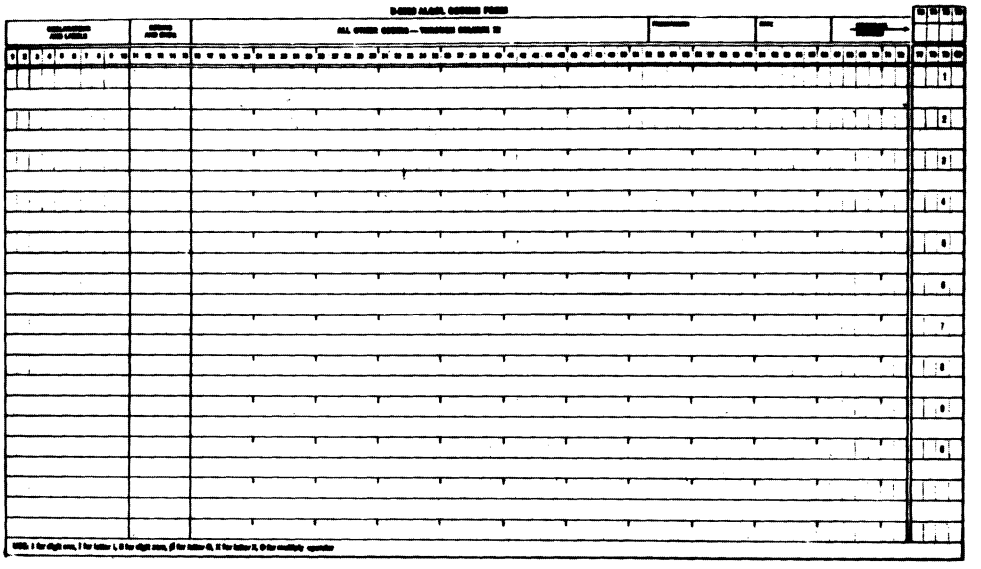
It is important to note the use of the separator semicolon in the above symbolic examples. The semicolon is used to separate declarations from declarations, declarations from statements and statements from statements. A semicolon is not needed immediately preceding an END, however.

A formal definition of a program is given at the end of Section 3.3.

At this point it is probably in order to digress briefly and discuss the format which one might follow in writing a program and how the program, once written, is communicated to the computer.

An ALGOL program is introduced into the computer as data for the compiler which subsequently produces an equivalent program in machine language. Normally this ALGOL source program is read into the computer from punched cards. These punched cards are prepared from the program as written by the programmer in long-hand. Each column of the punched card represents one character as written by the programmer.

To facilitate keypunching, as well as to provide the programmer with a suggested format to follow in writing his program, printed programming forms are often used. An example of such a form appears in the following figure:



A convention, which is often followed, is to begin only declarations and labels in column one; start only the bracket symbols BEGIN and END in column eleven; and begin all other constructs in column sixteen. The compiler will only recognize the first 72 columns. This leaves columns 73 to 80 for program identification and a card sequence number. The following example shows this format:

```
1           11          16                                  73
                        |                                   |
            BEGIN       |                                   |
                        |                                   | Ø RBIT001
REAL                    | A, W, MAX;                        | Ø RBIT002
INTEGER                 | IB, K;                            | Ø RBIT003
                        | S;                                | Ø RBIT004
                        | S;                                | Ø RBIT005
                        | S                                 | Ø RBIT006
            END         |                                   | Ø RBIT007
```

The programmer writes his program in ALGOL on a programming form such as the above. This form is transcribed, column by column, into punched cards. There will be a card created for each line written on the form. This deck of cards, the source deck, is then compiled and the program executed. As a by-product of compilation a listing of the source cards is produced. This usually constitutes the programmer's working document since it is always up to date. Adherence to a readable card format such as that used in the above example will produce readable listings, and the program deck is readily modified without any loss of readability.

It might be noted at this point that the compiler does not consider the information provided to it as a series of disjoint cards. Rather it treats the information as a continuous string of characters starting with the first column of a card, ending with column 72, and followed immediately by column one of the next card.

3.2 STATEMENTS AND DECLARATIONS

Statements and declarations are the building blocks from which an ALGOL program is constructed. They are, in a sense, analogous to sentences in the English language.

Statements are the active elements of an ALGOL program. They actually indicate some type of operation to be performed—add two numbers together, read in data, print a line, etc. Statements are normally executed sequentially, in the order in which they were written. This sequential flow may be altered, however, by a statement which indicates its successor to be other than the one immediately subsequent to it.

⟨statement⟩ ::= ⟨unconditional statement⟩|
⟨conditional statement⟩|⟨for statement⟩

The unconditional statement is similar to the imperative sentence in the English language in that it specifically states an operation to be performed. The conditional statement provides for the choice of alternative courses of action depending upon the answer to a particular question. The for statement allows for the repetition of a given process in a specifically controlled manner. The latter two types of statements will be discussed in Chapters 4 and 5.

Declarations differ from statements in that they are passive elements of the language. They are not really executed as are statements. The declaration serves to provide the ALGOL compiler with information about identifiers in the program. For example, a declaration is required to specify that the identifier for a particular variable is always to represent an integer.

⟨declaration⟩ ::= ⟨type declaration⟩ | ⟨array declaration⟩ |
⟨label declaration⟩ | ⟨file declaration⟩ |
⟨format declaration⟩ | ⟨list declaration⟩ |
⟨diagnostic declaration⟩ | ⟨procedure declaration⟩

These eight declarations provide the compiler with all of the information which it requires about a given program. The type declaration has already been encountered in Section 2.5. The other declarations will be discussed as the need arises.

A word of caution is in order at this point. All identifiers which a programmer chooses must be declared by one of the above declarations. This is, in fact, the way an explicit meaning is attached to a particular identifier. Thus, it is natural to group declarations above statements in a program according to the rules discussed in the following section.

Consider the following example of REAL and INTEGER declarations written in the previously discussed format.

1	11	16	73
	BEGIN		
REAL		X, ZBAR, T1, K;	
INTEGER		I, Q, KMAX1;	

These declarations say to consider the variables represented by the identifiers X, K, ZBAR, and T1 as REAL variables and those represented by I, Q, and KMAX1 as INTEGER variables.

3.3 THE BLOCK AND COMPOUND STATEMENTS

It has been shown that a statement has three possible forms. One of these is the unconditional statement defined as follows:

⟨unconditional statement⟩ ::= ⟨basic statement⟩ |
⟨compound statement⟩ | ⟨block⟩

The first of these, the basic statement, is a construct which stands alone in the program structure. Its many forms are specified by the following definitions:

⟨basic statement⟩ ::= ⟨unlabelled basic statement⟩ |
⟨label⟩: ⟨basic statement⟩

⟨unlabelled basic statement⟩ ::= ⟨assignment statement⟩ |
⟨go to statement⟩ |
⟨dummy statement⟩ |
⟨read statement⟩ |
⟨write statement⟩ |
⟨procedure statement⟩

For example, in ALGOL the process of adding the values of two variables A and B and assigning the sum to the variable X would be written as follows:

X ← A + B

This is an example of an assignment statement. The assignment statement as well as the other basic statements will be discussed in detail as the need arises.

A label provides a way of uniquely identifying a particular statement. Labels and their use are discussed in Section 3.5.

The other two forms of the unconditional statement, the compound statement and the block, require formal definition and discussion.

The student may not fully appreciate all aspects of the concepts developed in the remainder of this section. However, due to the importance of these concepts it will be necessary to continually review them as further material is developed.

⟨compound statement⟩ ::= ⟨unlabelled compound statement⟩ |
⟨label⟩: ⟨compound statement⟩

⟨unlabelled compound statement⟩ ::= BEGIN ⟨compound tail⟩

⟨compound tail⟩ ::= ⟨statement⟩ END | ⟨statement⟩;
⟨compound tail⟩

The above syntactical definitions state that an unlabelled compound statement is composed of the bracket symbol BEGIN, followed by one or more statements separated by semicolons and terminated by END which is also a bracket symbol. Symbolically it could be represented by the following example:

BEGIN S; S; S; S END

It is understood, of course, that any number of statements may appear between the bracket symbols. Note that no semicolon need appear immediately preceding the END.

The purpose of the compound statement is to group together several basic statements which are to be considered, in some sense, as a logical entity. It would be used, for example, in a situation where three assignment statements are to be executed if the answer to a given question is true and all three bypassed if the answer is false. In this case, the three assignment statements are considered as a logical group. Such a compound statement might be written as shown below:

```
          BEGIN                                    EX.   1
                    X ← A + B;                     EX.   2
                    Y ← C;                         EX.   3
                    Z ← X - Y                      EX.   4
          END                                      EX.   5
```

The purpose of the compound statement will be fully appreciated only when the other types of statements, the conditional statement and the for statements have been discussed.

A block differs from a compound statement in the important respect that it contains not only statements but also declarations between the bracket symbols. These declarations must appear immediately following the BEGIN and preceding any statements. The formal definition of a block is:

⟨block⟩ ::= ⟨unlabelled block⟩ | ⟨label⟩ : ⟨block⟩

⟨unlabelled block⟩ ::= ⟨block head⟩ ; ⟨compound tail⟩

That is, an unlabelled block is defined as a block head followed by a semicolon and a compound tail.

Since we have already discussed the compound tail we need only define a block head.

⟨block head⟩ ::= BEGIN ⟨declaration⟩ | ⟨block head⟩ ; ⟨declaration⟩

Symbolically we can represent a block which might contain two declarations and four statements.

BEGIN D; D; S; S; S; S END

The block head is represented by BEGIN D; D and the compound tail by S; S; S; S END with the two parts separated by a semicolon as defined. Consider an example of a block written in ALGOL. This example illustrates the structure of a block but has no realistic meaning.

```
          BEGIN                                    DECL  01
REAL                X, Y;                          DECL  02
INTEGER             J, K;                          DECL  03
                    X ← Y;                         DECL  04
                    Q ← J + K                      DECL  05
          END                                      DECL  06
```

One purpose of a block is to introduce a new level of nomenclature. An identifier has the same meaning inside a block as it has outside as long as it is not redeclared within the block head. Q in the above example is such an identifier. Thus Q is said to be global to this block. If it is declared in the block head of some block, the meaning within that block may be completely different from its meaning outside the block. The new meaning holds while the block is executed and then reverts back to the meaning in effect outside the block. Identifiers are said to be local to the block in which they are declared. The local nature of identifiers used as labels is discussed further in Section 3.5.

To fully understand the above paragraph we must go back to the definition of a statement as a block. This means that syntactically a block may occur as a statement within a program. In other words, a block may be nested within another block or a compound statement and logically considered as a statement. For example, a program might be diagrammed as follows:

BEGIN D; D; S; S; S END.

However, when the program is diagrammed more completely, one or more of the statements may be a block.

BEGIN D; D; S; BEGIN D; S; S END; S END.

The underlined portion is, therefore, a block nested within a program. We will reserve a discussion of the full power provided by the block concept until Chapter 9.

The compound statement can occur in much the same way. If a program were diagrammed in detail, for example, one or more of the statements might be compound.

BEGIN D; D; S; BEGIN S; S; S END; S END.

The underlined portion represents the compound statement. The compound statement differs from the block in that it contains no declarations and therefore no change in nomenclature.

A formal definition of a program can now be given.

⟨program⟩ : : = ⟨unlabelled block⟩.|⟨unlabelled compound statement⟩.

Note the explicit appearance of a period as the final delimiter of a program. By far, the majority of programs are written as unlabelled blocks.

3.4 ASSIGNMENT STATEMENTS

The assignment statement is the first unlabelled basic statement which will be discussed in detail. It is the fundamental statement in the language and also the most straightforward to understand and use.

Basically the assignment statement causes the value of an expression to be assigned to a variable appearing to the left of a←. The← is called the replacement operator and means that the value of the expression to the right of the ← is assigned to the variable appearing on the left of the←. The ← indicates actual replacement rather than equality and so the previous value of the variable is lost.

Values assigned to variables are "stored" until one of the following occurs.

1. A new value is provided by an assignment statement.
2. A new value is provided by a read statement (to be covered in Section 3.7).
3. The block in which the variable is declared is exited.

Since most programs depend upon evaluating expressions and assigning the resultant value to other variables, the assignment statement is the work horse of the language. Consider its formal definition.

⟨assignment statement⟩ ::= ⟨left part list⟩⟨arithmetic expression⟩
⟨left part list⟩ ::= ⟨left part⟩ | ⟨left part list⟩⟨left part⟩
⟨left part⟩ ::= ⟨variable⟩←

Since the arithmetic expression was covered in Section 2.7, an understanding of the assignment statement requires only a discussion of the idea of a left part list and a left part. A left part is defined as a variable followed by the replacement operator.

A ←

LAMDA ←

X1 ←

The above are examples of the left part list. The left part list may contain left parts as indicated in the following examples:

A←LAMDA←XI←

I←J←K←

The left part list allows the value of an expression to be assigned to several variables in the same statement. A restriction on this type of assignment statement is that all variables appearing in the left part list must be of the same type (real or integer).

The execution of an assignment statement may be considered as two steps:

1. The expression on the right of the ← is evaluated.
2. The value of the expression is assigned to all of the variables in the left part list.

There can be a difference between the type of the arithmetic expression and the type of the variables in the left part list. Two different cases arise:

1. If the left part list is of type REAL, and if the value of the arithmetic expression is of the type INTEGER, the value is assigned unchanged.
2. If the left part list is of type INTEGER, and if the value of the the arithmetic expression is of type REAL, the function ENTIER (AE+0.5), where AE represents the value of the arithmetic expression, is automatically applied. This, therefore, causes the value to be rounded to an integer before it is assigned.

The following examples will indicate the form of the assignment statement.

```
X ← A × (B - C)
ALPHA ← BETA ← GAMMA ← 3.1416
XI ← R × COS(THETA)
FUNCTIONOFX ← A × X*3 + B × X*2 + C × X + D
XSUM ← XSUM + SQRT (X)
```

Two final points can be made regarding the assignment statement. As is indicated by the syntactical definition only variables may appear in the left part list. Therefore, a construct of the form X+Y←B is obviously invalid.

The second point involves the use of the same variable in the left part list and in the arithmetic expression. For example, X ← X+1. When this assignment statement is executed the current value of X will be used in evaluating the expression and that value will be assigned to X. In this example, the value of X will be one greater after the statement is executed.

3.5 GO TO STATEMENT AND LABELS

It was said previously that statements were executed sequentially, in the order in which they appear, unless a statement is encountered which specifically states that the next statement to be executed is not the next one in sequence. This ability to change the sequential flow of the program logic contributes in a major way to the power of a digital computer.

One way in which the sequence of execution of statements can be changed is by the use of the go to statement. This statement specifically designates the statement which is to be executed next.

⟨go to statement⟩ ::= GO TO ⟨label⟩

From its formal definition it is seen that the go to statement is made up of the reserved words GO TO and a label which is defined below.

It is apparent that to effect a change of control from one statement to another, not in sequence, there must be a way of uniquely designating the destination statement. This is dore by the use of a label. Formally a label is defined as an identifier. Therefore, a label may be arbitrarily chosen within the restriction imposed upon the choice of characters making up an identifier. The programmer serves notice that a given identifier is used as a label by means of a label declaration, which is written similar to type declarations.

⟨label declaration⟩ ::= LABEL ⟨label list⟩
⟨label list⟩ ::= ⟨label⟩ | ⟨label⟩, ⟨label list⟩
⟨label⟩ ::= ⟨identifier⟩

This declaration must occur in the block head of any block that contains a labelled statement. The label list contains only those identifiers used to label statements in this block. It is only within this block, and any blocks nested within it, that these labels may appear in a go to statement. Thus, a go to statement may not reference a label in a block nested within the block which contains the go to statement.

Recall that the definition of a basic statement included as an alternative definition a ⟨label⟩:⟨basic statement⟩. Also, a block could be of the form ⟨label⟩:⟨block⟩ and a compound statement of the form ⟨label⟩:⟨compound statement⟩. It is apparent then, that any construct which may be considered as a statement may be uniquely labelled by preceding it with a label followed by a colon. We can effect a change of control to any statement by writing GO TO ⟨label⟩, where the label precedes the statement which is to be executed next. For example, GO TO OVER would cause the statement with the associated label OVER to be executed next. The following section of ALGOL source coding will serve as an example of both assignment statements and go to statements. This example is not a very practical one since it is obvious that once the statement labelled START is encountered the program will be in a never ending loop.

```
          BEGIN                                          009
LABEL                 START;                             010
START:                X ← Y;                             011
                      A ← CØS (THETA);                   012
                      GØ TØ START                        013
```

An occasionally useful statement is the dummy statement which we define as empty.

⟨dummy statement⟩ ::= ⟨empty⟩
⟨empty⟩ ::= {the null string of symbols}

This statement represents no actual operation but serves to place a label in a convenient location. For example, it allows the delimiter END to be labelled, and, if a transfer of control to the label preceding the END is specified, a block or compound statement can be executed without executing all of its statements. Note that the label L1, since it puts a dummy statement before the END, makes the semicolon mandatory after the statement C ← C + 1.

```
          BEGIN                                              10
LABEL                L1,FLØW;                                11
                     X ← A +B;                               12
                     GØ TØ L1;                               13
FLØW:                A ← SQRT (C + D);                       14
                     C ← C + 1;                              15
L1:       END                                                16
```

3.6 THE COMMENT

Often the purpose of a section of an ALGOL program can be made more apparent by including a comment on the program at an appropriate point within the sequence of statements or declarations. A comment may be included in the program by using the reserved word COMMENT in one of the following ways:

; COMMENT { any sequence of characters not containing ; } ;
BEGIN COMMENT { any sequence of characters not containing ; } ;

The first of these says that after any semicolon which separates statements and declarations, as previously discussed, the reserved word COMMENT may be written followed by any comment written in the legitimate character set except the semicolon. The semicolon serves to indicate the end of a comment. The second case indicates that a comment may follow a BEGIN.

Another form of the comment, not using COMMENT, may follow an END as indicated below:

END { any sequence of characters not containing END or ELSE or WHILE or UNTIL or any special character unless it occurs as a part of an ALGOL number }

This is useful in making commentary at the end of a compound statement, a block, or a program. It should be emphasized that these comments are for documentation purposes only. They appear on the program listing which is produced by the compiler but have no effect on the machine language generated.

```
          BEGIN
CØMMENT   THIS EXAMPLE ILLUSTRATES THE
          USE OF CØMMENTARY WITHIN A
          CØMPØUND STATEMENT;
          X ←Y;
          A ←B + C + D;
CØMMENT   SØ DØES THIS;
          I ← J ← K← 0;
          PHI ← THETA
          ØF THE EXAMPLE
END
```

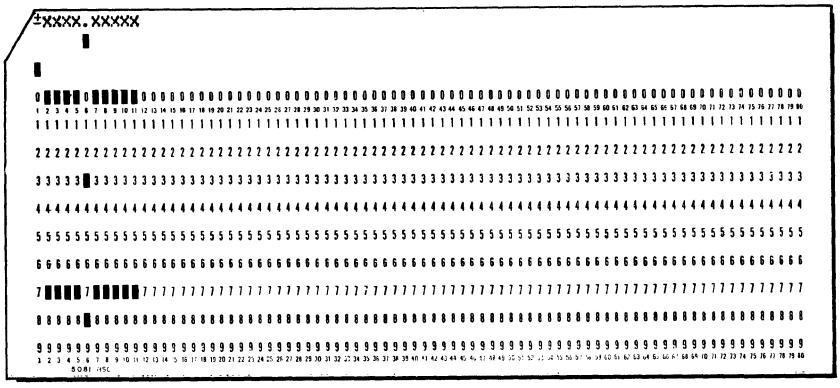
3.7 SIMPLIFIED INPUT/OUTPUT

With the discussion of one additional topic, enough of the ALGOL language will have been developed to write simple but complete programs. This additional topic is input/output—how to get data into the computer for the program to operate on and how to get the results back out to the programmer. The subject will be treated very briefly here, to allow complete programs to be written, and developed in depth in Chapter 7.

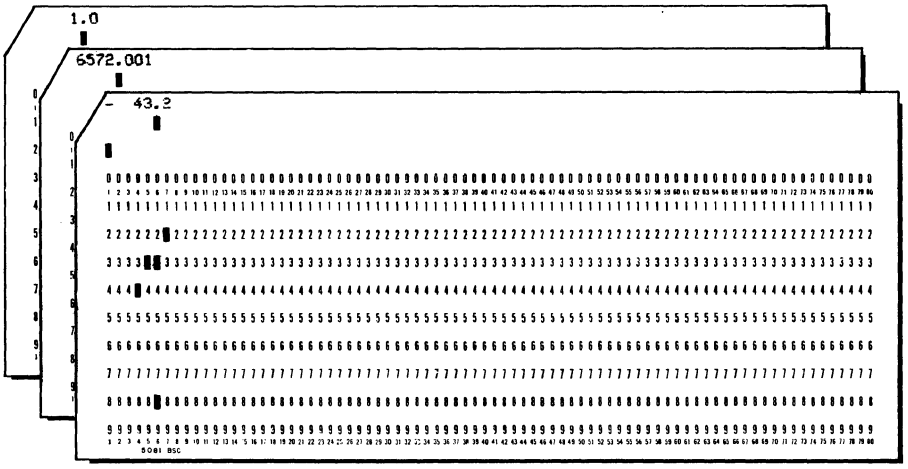
The media available for input/output are fairly extensive, including magnetic tape, paper tape, punched cards, printed hard copy, etc. Furthermore, with developments in optical character recognition, new forms of input/output will be developed. We choose to consider only the most common of these possible media, punched cards for input and the printed line for output. The reasons for this choice are twofold. First, these two media are sufficient for many scientific applications. Secondly, this allows the basic concepts of data input/output to be developed. These concepts can be easily extended to other input/output devices by the student, especially since he will have the capability to read the reference documents which use the metalinguistic approach to definition.

To provide a basic input capability, consider the data as it would appear on a punched card.

Data will be, for now, punched into cards, one value per card. The number will be punched as in the example below where the symbol X represents any digit. The largest value which can be read in will be 9999.999999. The sign must appear in column 1 of the card and the decimal point in column 6. If the number is positive, column 1 may be left blank, implying a plus sign.



The following cards represent the input values -43.2, + 6572.001, 1.0.



The format which will be used in communicating results from the computer to the programmer on the printer is shown below:

```
THIS IS A SAMPLE OF PRINTED OUTPUT.

±XXXX.XXXXXX  ±XXXX.XXXXXX  -----  -----  -----  -----
±XXXX.XXXXXX  ±XXXX.XXXXXX  -----  -----  -----  -----
--------------  --------------  -----  -----  -----  -----
--------------  --------------  -----  -----  -----  -----
```

It will be possible to print a line of heading information followed by a blank line and the data. The data will appear in a tabular form with from one to six values on a single line.

To accomplish the basic input and output functions which we have defined above, it will be necessary to provide several ALGOL constructs which will be discussed fully in Chapter 7. These constructs are in fact declarations and as such, must appear in some block head.

At this point these declarations will be defined as a series of five cards which should be inserted immediately following the initial BEGIN in the program.

```
          BEGIN                                          01
FILE IN                CARD (2, 10);                     02
FILE ØUT               LINE 1 (2, 15);                   03
FØRMAT IN              FIN (F12.6);                      04
FØRMAT ØUT             FT1 (X10,"----------"//);         05
FØRMAT ØUT             FT2 (6 (X4, F12.6, X4) /);        06
```

The form of a program will be, therefore, as indicated in the above example. The BEGIN which indicates the start of a program would appear as card number 01 in the example. Cards numbered 02 through 06 are those necessary to set up the basic input/output capability. Card number 05 deserves special consideration. The heading information on the printed output page is the string of characters which appear between the quote marks on this card. This string may contain any of the defined characters except the quote itself and for the present must be contained entirely on one card.

To read data which is punched into cards according to the format defined above, the read statement is used.

READ (CARD, FIN, ⟨list⟩)[⟨label⟩]

The list is a series of identifiers, separated by commas, which represents those variables which are to be assigned the values read from the cards.

As many cards will be read as there are identifiers in the list for each read statement executed. It is important, then, that there exists a one-to-one correspondence between the elements of the list and the cards to be read.

Notice the label appearing in square brackets. This entire construct (including the brackets) is optional. Its function is to provide a label which uniquely specifies a statement to which control should be transferred if an attempt is made to read when no more data cards exist. This is useful when a program is written to do the same operations on successive sets of data. A data set can be read, the computation performed, and control transferred (via a go to statement) back to the read statement to read in the next set of data. When no more data exists the next statement executed will be the one preceded by the label.

READ (CARD, FIN, A, B, C, ALPHA, PHI) [THROUGH]

The above read statement will read in five cards assigning the successive values, in order, to the variables, A, B, C, ALPH, PHI. If an attempt is made to execute this statement again, after all data has been read, control will be transferred to the statement labelled THROUGH.

The write statement is highly analogous to the read statement.

WRITE (LINE, FT1)

Execution of this statement causes the heading information to be written on the printer.

WRITE (LINE,FT2, ⟨list⟩)

Execution of the above statement will effect the printing of the values of the variables or expressions represented by the elements in the list, six numbers per line, until the list has been satisfied. Fewer than six numbers may be printed on the final line if there is not a multiple of six elements in the list.

WRITE (LINE, FT2, LAMDA, I, J, 314159, A + B)

This example would cause the values of the five list elements to be printed. Notice that arithmetic expressions may appear as elements in the list of a write statement. They will be evaluated and their value printed.

The following example utilizes those constructs which have been developed to express the previously discussed algorithm for solving simultaneous linear equations as an ALGOL program. Recall that the values for x and y may be formed from the following relationships:

$$x = \frac{ec - bf}{ae - db} \qquad y = \frac{af - dc}{ae - db}$$

```
          BEGIN                                                                  SLEQ  1
FILE IN         CARD(2,10);                                                      SLEQ  2
FILE OUT        LINE 1 (2,15);                                                   SLEQ  3
FORMAT IN       FIN(F12.6);                                                      SLEQ  4
FORMAT OUT      FT1(X10,"SOLUTION OF SIMULTANEOUS LINEAR EQUATIONS"//);          SLEQ  5
FORMAT OUT      FT2(6(X4,F12.6,X4)/);                                            SLEQ  6
REAL            X,Y,A,B,C,D,E,F,T;                                               SLEQ  7
LABEL           AGAIN,FINISHED;                                                  SLEQ  8
                WRITE(LINE,FT1);                                                 SLEQ  9
AGAIN:          READ(CARD,FIN,A,B,C,D,E,F)[FINISHED] ;                           SLEQ 10
                WRITE(LINE,FT2,A,B,C,D,E,F);                                     SLEQ 11
                T←A×E-D×B;                                                       SLEQ 12
                X← (E×C-B×F)/T ;                                                 SLEQ 13
                Y← (A×F-D×C)/T ;                                                 SLEQ 14
                WRITE(LINE,FT2,X,Y);                                             SLEQ 15
                GO TO AGAIN;                                                     SLEQ 16
FINISHED: END.                                                                   SLEQ 17
```

This sample program is complete in every detail and, if punched into cards, would compile and run on a B 5500 computer. Notice that the program starts with BEGIN and the five declarations which will be used to provide the basic input/output capability. The sequence of characters between the quotation marks in the declaration on card SLEQ 5 will be printed as the heading information.

In addition to the five declarations which are used to provide the input/output capability, nine real variables and two labels are declared.

The first write statement effects the printing of a heading on the output. The heading will say SOLUTION OF SIMULTANEOUS LINEAR EQUATIONS. The read statement bearing the label AGAIN will cause the variables, A, B, C, D, E, and F to be initialized to the values read from six consecutive punched cards. The write statement following the read will print out the input data. This is always good programming practice.

After the calculations are performed the results will be printed out by the second write statement. Control will then be transferred back to the read statement via the GO TO AGAIN construct. This loop will be repeated until no more data sets exist at which time control will be transferred to the label, FINISHED, specified in the action label part of the read statement and the program will be completed.

EXERCISES

1. Write ALGOL assignment statements for the following mathematical formulas:

   a. $y = a - bx + cx^2 - dx^3 + ex^4 - fx^5$

   b. circle area $= \pi \times (\text{radius})^2$

   c. gross pay $= 30 \times (\text{normal rate}) + [(\text{total hours}) - 30] \times (\text{overtime rate})$

   d. discriminant $= b^2 - 4ac$

   e. tangent of $x = \dfrac{\sin x}{\cos x}$

   f. $\pi = 3.141593$

   g. new distance $=$ (starting distance) $- \dfrac{(\text{speed}) \times (\text{time})}{(\text{route tortuosity})}$

2. Find the syntactical error in each of the statements:

   a. W2F← (ALPHA - BETA)5@-13

   b. PLUTO: A ← B4U + ACT*-Q

   c. WRONG: 60 ←MIN

   d. F ← TIZ + 136. - A/2 +B64000

   e. 43: X ← X + .195

   f. GOTO LOST

   g. L1:L2: T ← T - 2DELTEMPAVE

   h. GO TO 12

   i. BEGIN LABEL EXT; EXIT: END

   j. BEGIN Y ← Y + 1 , X ← X - 1 END

   k. J ← Z + 6.23(X-Y)

   l. XZ ← YZ ← A-B←Z←4

   m. PQY← -4.3 x (H×(X-Y)

   n. A ← B@4 + Z*(I-4)

   o. GO TO L+1

p.
```
        BEGIN
        FILE IN CARD (2, 10);
        INTEGER A, B, D;
        REAL I, C, E;
        C← 3; E ← 4.67;
        LABEL FINIS;
        READ(CARD,..... );
FINIS:  END
```

3. The following compound statement represents the most efficient way to compute Z, given A.

```
BEGIN
        TEMP1 ← 2.5 X A +4; TEMP2 ← TEMP1 *2;
        Z ←(TEMP1+(TEMP2 + 1)/A)/(TEMP2-1)
END;
```

Write an equivalent single assignment statement for computing Z from A.

4. Write a program to read single data values from an unknown number of cards, print each value, and accumulate the sum of the values and the sum of their squares. Compute and print the average and the standard deviation of the complete set of numbers. Recall that, for N values, (ave.) = (sum of values)/N and $(\text{std. dev.})^2 = [N \times (\text{sum of squares}) - (\text{sum of values})^2]/(N(N-1))$.

5. Write a program to read an angle, x, from a data card, compute values for sin(x), tan(x), $\log_{10}(\sin(x))$, and $\log_{10}(\tan(x))$ and print them on a single line. Repeat this process for as many data cards as are presented to the program, thus printing a table. Compare the results with values tabulated in a handbook. Assume the angles are in degrees.

CHAPTER 4 - CONDITIONAL STATEMENTS

4.1 THE CONCEPT

One of the features of the digital computer that distinguishes it from lesser devices, such as the desk calculator, is its decision-making capability. Although widely misunderstood in degree, it is this capability, along with the storage (or memory) capability, that allows computers to solve complicated problems. In reality, the computer itself only makes binary (yes/no) decisions. It is the combined effect of many binary decisions, in a program, that seems to endow a computer with a rudimentary intelligence. Note that it is the program, written by humans, which embodies the intelligence, if any, exhibited by the computer.

The binary decision capability of computers takes the form of alternative actions depending on the result of very simple comparisons. For example, most computer hardware is capable of sensing whether a given numeric value is zero or non-zero and whether its sign is negative or positive. If the problem decision to be made is at all complex, the difficulty of correctly programming it as a series of binary decisions in machine language can be very great.

ALGOL greatly simplifies the programming of decision processes. It does this by allowing the decisions to be expressed in a highly readable form. For example, a special action to be taken, in case the value represented by a variable ALTITUDE is negative, might be expressed by the ALGOL statement:

```
IF ALTITUDE < 0 THEN GO TO CRASHED
```

A considerably more complex decision process is found in the example,

```
IF (A>B OR C = 0) AND D/E ≤ B × A
THEN X ← Y ELSE Y ← X/2
```

Nevertheless, the latter example is still simple to write, to read and to modify (when necessary). Either example is easily converted into correct and efficient machine language by the ALGOL compiler.

One purpose of the present chapter is to set forth the rules governing the writing of statements such as those above. These are called conditional statements. Their syntax will be discussed in Sections 4.3 and 4.4, after development of one component, the Boolean expression, is completed in Section 4.2.

4.2 BOOLEAN EXPRESSIONS

A vital portion of all conditional statements is the part which expresses the truth value upon which the condition is based. Mathematically, these are called Boolean expressions. For simplicity, the present subset of ALGOL utilizes these only in conditional statements. Further, the subset narrows the definition of Boolean expression to include only the most needed forms.

Analogous to the primary in arithmetic expressions, there is the Boolean primary. It is defined as

⟨Boolean primary⟩ ::= ⟨relation⟩ | (⟨Boolean expression⟩)

The relation is the basic truth value, which is either true or false. It is formed according to the formulas,

⟨relation⟩ ::= ⟨arithmetic expression⟩ ⟨relational operator⟩ ⟨arithmetic expression⟩

⟨relational operator⟩ ::= < | ≤ | = | ≠ | ≥ | >

The relational operators have the conventional mathematical meanings, as follows:

<	is less than
≤	is less than or equal to
=	is equal to
≥	is greater than or equal to
>	is greater than
≠	is not equal to

They assert a relationship between two arithmetic expressions which may be either true or false, depending on the numerical values of the variables or numbers that occur in the arithmetic expressions.

Note that literally any arithmetic expression is allowed on either side of a relation. Thus, N≠-7 and 2=2 are legal relations, as is K * SIN(BETA) < (P * COS(PHI))/(1+N). Note that these are also perfectly good Boolean primaries.

A word of caution is in order on the use of REAL arithmetic expressions in relations. Most decimal fractions cannot be represented exactly as a binary (base 2) number. For example, if the value of A is computed by originally setting it to zero and then successively adding 0.1, the relation A = 1.0 never becomes true. The situation is analogous to that experienced in decimal (base 10) calculations where 3 * (1/3) ≠ 1 because 1/3 cannot be represented exactly as a decimal fraction. Note that (1/3 + 1/3 + 1/3) < 1 if 1/3 is approximated by 0.3333333333, and (2/3 + 2/3 + 2/3) > 2 if 2/3 is approximated by 0.6666666667. The most prudent rule is to avoid the use of the relations which involve REAL arithmetic values where the case of exact equality is important.

The two logical operators AND and OR are needed in forming Boolean expressions. The operator AND gives the logical product and OR gives the logical sum of two truth values, according to the following table, where BE1 and BE2 represent Boolean expressions:

BE1	False	True	False	True
BE2	False	False	True	True
BE1 AND BE2	False	False	False	True
BE1 OR BE2	False	True	True	True

We are now adequately prepared to define the Boolean expression. For completeness, the formula for a Boolean primary is repeated here.

⟨Boolean expression⟩ ::= ⟨Boolean factor⟩ | ⟨Boolean expression⟩ OR ⟨Boolean factor⟩

⟨Boolean factor⟩ ::= ⟨Boolean primary⟩ | ⟨Boolean factor⟩AND ⟨Boolean primary⟩

⟨Boolean primary⟩ ::= ⟨relation⟩ | (⟨Boolean expression⟩)

Note the roles played by the two logical operators. The AND operator has a higher precedence than does OR. This causes the Boolean factor to be treated as an entity just as the term A/B is an entity in the arithmetic expression M+A/B. For example, the Boolean expression X ≠ 1 OR Y = 0 AND Z ≠ 0 is evaluated by the steps:

1. Obtain the truth values for all the relations

2. Evaluate the Boolean factor Y = 0 AND $Z \neq 0$

3. Combine the result with the truth value for $X \neq 1$, via the OR operator

Since a Boolean expression can be made a Boolean primary by enclosing it in parentheses, the order of precedence of the logical operators can be overridden. This is analogous to the use of parentheses in arithmetic expressions.

The student should verify the classifications of examples given below:

Legal	Illegal
Boolean primaries	
-1 = 0	A > B AND C < D
1-A > B* (-E)	$A \neq B$ OR C = D
(X=Y OR W-K<4)	1-W*2
($F \neq N$ AND 1+N≥A/D)	
Boolean factors	
X = 0 AND $Y \neq 0$	$A \neq B$ OR C = D
A > 1 AND (B = 0 OR C < D)	1 + A AND Z>0
(A=B OR C=D) AND (X<2 OR Y<2)	
$F \neq N$ AND 1 + N≥A/D	
Boolean expressions	
$A \neq B$	(B*2-4 x A x C)
DX > DXMAX OR DY > DYMAX	
I = 0 AND J = 0 OR K≥1	
(I $\neq$ 0 OR J $\neq$ 0) AND (A>0 OR X/Y<1)	

4.3 IF STATEMENT

The most commonly used conditional statement is the if statement. (A complete definition of conditional statements is given in Section 4.4).

⟨if statement⟩ : := ⟨if clause⟩ ⟨unconditional statement⟩

We see that any if statement is composed of two parts. Recall that the unconditional statement was defined and discussed in Chapter 3.

⟨if clause⟩ : := IF ⟨Boolean expression⟩THEN

This requires that all if clauses begin with the reserved word IF and end with another reserved word THEN. Recall in Section 2.2 that IF and THEN are formally called sequential operators.

Note that, although the if clause seems to have a very simple form, the occurrence of the Boolean expression within it adds all of the variety brought out in the preceding section.

So, an if statement may take on great simplicity, such as:

```
IF N≠M THEN GO TO RECT
```

or

```
IF A< L THEN B ←2
```

On the other hand, either the Boolean expression or the unconditional statement or both can be quite complex. Since the latter may be a block, an if statement can constitute a major portion of an ALGOL program.

There are no reservations on the kinds of unconditional statements that can occur in if statements. For example:

```
IF Z/N > 2 THEN BANK: Z← ZLIM
```

is perfectly legal because

```
BANK: Z ←ZLIM
```

is a legitimate basic statement, one of three forms of unconditional statements, defined in Chapter 3. In the same way, it is legal to write

```
IF A < B OR C < B THEN

UNDER: BEGIN D ← - D; K ← S ← 0 END
```

More examples of if statements are given at the end of Section 4.4.

The meaning of an if statement is not hard to explain or to remember because it coincides with normal English usage. That is, if the truth value of the Boolean expression is true, the unconditional

statement is executed; if not, no action is taken beyond evaluation of the Boolean expression. If the truth value is false, control passes immediately to the statement following the if statement. If the truth value is true, the unconditional statement may cause control to be transferred to another portion of the program. If that does not happen, the statement following the if statement is executed next.

It is legal to refer to (via a GO TO) any label within an if statement from points outside that if statement. Control then does not depend on the Boolean expression in the if clause of that if statement.

4.4 CONDITIONAL STATEMENTS

We now have sufficient preparation to discuss the complete definition of a conditional statement:

⟨conditional statement⟩ ::= ⟨if statement⟩ | ⟨if statement⟩ELSE ⟨statement⟩ | ⟨if clause⟩⟨for statement⟩ | ⟨label⟩:⟨conditional statement⟩

For convenience, we repeat previously given definitions of four of the above constituents.

⟨if statement⟩ ::= ⟨if clause⟩⟨unconditional statement⟩

⟨if clause⟩ ::= IF ⟨Boolean expression⟩ THEN

⟨unconditional statement⟩ ::= ⟨compound statement⟩ | ⟨basic statement⟩ | ⟨block⟩

⟨statement⟩ ::= ⟨unconditional statement⟩ | ⟨conditional statement⟩ | ⟨for statement⟩

A fifth major constituent, the for statement, is to be taken up separately in Chapter 5.

The above definition of a conditional statement implies to us the following:

1. The if statement, discussed in the previous section, is only one form of a conditional statement.

2. Another form, yet to be discussed, appends to an if statement the reserved word ELSE followed by a statement. No semicolon appears before the sequential operator ELSE because ELSE can not be the beginning of a new statement.

3. Another form is an if clause followed by a for statement, to be discussed in Chapter 5.

4. A conditional statement can be labelled (with one or more labels) or it may be unlabelled.

Before taking up the ELSE option, it is worth contemplating the recursiveness of the definition of the conditional statement. The power of the metalinguistic notation is well displayed here. How else could we as easily establish that it is legal to write, among countless other forms, a single conditional statement of the form:

IF ⟨Boolean expression⟩ THEN
⟨basic statement⟩ ELSE

IF ⟨Boolean expression⟩ THEN
⟨label⟩ :⟨basic statement⟩ ELSE

IF ⟨Boolean expression⟩ THEN
⟨compound statement⟩ ELSE

⟨label⟩: IF ⟨Boolean expression⟩ THEN
⟨basic statement⟩ ELSE
⟨for statement⟩

Recall that such an example is not meant to demonstrate how to write (a portion of) an ALGOL program. It illustrates only that, if such a construct were to occur naturally in the writing of a program, it would be syntactically correct and meaningful to the ALGOL compiler.

The form of the conditional statement that employs the ELSE is sometimes called the "IF...THEN...ELSE statement." This arises from the definitions of its constituents, since

⟨if statement⟩ELSE ⟨statement⟩

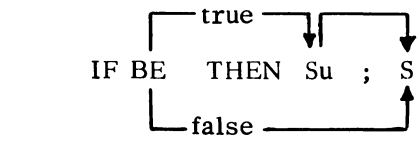
could equally well be written

IF ⟨Boolean expression⟩ THEN ⟨unconditional statement⟩ELSE
⟨statement⟩

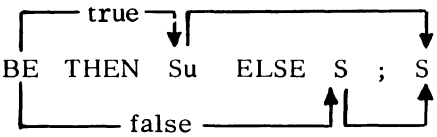
Again, the meaning may be correctly surmised from English usage. If the truth value is true, the unconditional statement portion is executed. If it is false, the statement (following the ELSE) is executed.

Control, unless directed elsewhere via a GO TO, passes to the next statement. The passage of control, in the absence of any GO TO, is perhaps better defined by the diagrams:

IF statement

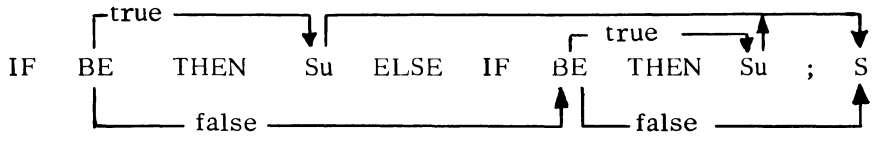


IF...THEN...ELSE statement

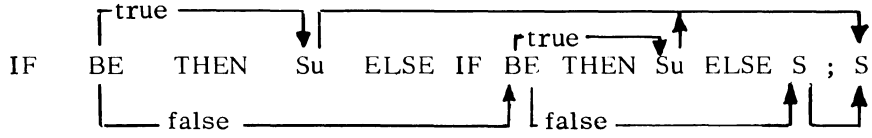


Here BE represents Boolean expression, Su represents unconditional statement, and S represents statement. Note the absence of a semicolon preceding ELSE.

In case the statement following the ELSE happens to be an if statement, the passage of control, without any GO TO, is diagrammed as



Another case might be



In the most general case, a conditional statement can be a series of conditions and the evaluation continues until a truth value of true is found. When this occurs, the next succeeding unconditional statement is executed. If none of the Boolean expressions has a truth value of true, the statement following the rightmost ELSE is executed. If no ELSE appears after the rightmost THEN, control continues in sequence.

A go to statement may lead to a label within a conditional statement. If this occurs, the flow from that point is determined by the same rules as if that point had been reached by entry through the if clause at the front of the conditional statement.

Examples are given below for the new constructs discussed in this section. These include, for completeness, and later reference, an example of the ⟨if clause⟩⟨for statement⟩ construct, although the for statement is not discussed until Chapter 5.

EXAMPLES

If clauses

IF A > B THEN

IF A ≠ C OR X < Y+1 THEN

IF X > 0 AND Y < 0 THEN

If statements

IF A > B THEN A ← A -1

IF W < 0 THEN BEGIN W ← 0 ; GO TO L1 END

IF X ≥ 0 AND Y ≥ 0 THEN GO TO QUAD

Conditional statements

IF A < B THEN A ← A + 1

IF E > L THEN Z ← Y ELSE Y ← Z - E

```
SCALE:  IF S < 10.1 THEN S ← 10 ELSE
        IF S < 20.2 THEN S ← 20 ELSE
        IF S < 40.4 THEN S ← 40 ELSE
        IF S < 80.8 THEN S ← 80 ELSE
        BEGIN S ← S/10; GO TO SCALE END
IF N > 0 THEN FOR I ← 1 STEP 1 UNTIL N DO
   H [I] ← A + B X T [I] + C X T [I] * 2
```

The following problem will serve as an example of the use of conditional statements in a program. The program evaluates e^X from the infinite series approximation using the first 20 terms of the series.

Sample Problem: $e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$

```
          BEGIN                                                          ETOX  1
COMMENT         EVALUATE E*X  FROM SERIES APPROXIMATION;                  ETOX  2
FILE IN         CARD(1,10);                                              ETOX  3
FILE OUT        LINE 1 (2,15);                                           ETOX  4
FORMAT IN       FIN(F12.6);                                              ETOX  5
FORMAT OUT      FT1(X10,"E*X = 1+X/1+X*2/2+X*3/6+... R.E.JOHNSON"//);    ETOX  6
FORMAT OUT      FT2(6(X4,F12.6,X4)/);                                    ETOX  7
REAL            X,ETOX,ETOXT;                                            ETOX  8
INTEGER         I;                                                       ETOX  9
LABEL           FIRST,SECOND,LAST;                                       ETOX 10
                WRITE(LINE,FT1);                                         ETOX 11
FIRST:          READ(CARD,FIN,X)[LAST];                                  ETOX 12
                I ← 2;                                                   ETOX 13
                ETOXT ← X;                                               ETOX 14
                ETOX ← X;                                                ETOX 15
SECOND:         IF I ≤ 19 THEN                                           ETOX 16
          BEGIN                                                          ETOX 17
                ETOXT ← ETOXT × X/I;                                     ETOX 18
                ETOX ← ETOX + ETOXT;                                     ETOX 19
                I ← I + 1;                                               ETOX 20
                GO TO SECOND                                             ETOX 21
          END;                                                           ETOX 22
                ETOX ←1+ETOX;                                            ETOX 23
                WRITE(LINE,FT2,ETOX,EXP(X));                             ETOX 24
                GO TO FIRST;                                             ETOX 25
LAST:     END  OF PROGRAM.                                               ETOX 26
```

EXERCISES

1. What is the value of X after the following is executed? Assume A = 6.3, B = 1.2, C = D = 7.

   If A > B or C $\neq$ 0 and D $\leq$ 3 then X $\leftarrow$ 1 else X $\leftarrow$ 23

2. Write conditional statements that correspond to the following verbal statements. If the statement is true, then transfer to label MRE otherwise transfer to label FIN.

   a. Either Q exceeds A, or it exceeds B, or it exceeds C, or is less than D.

   b. Both LS and MO must lie between 1.5 and 8.6.

   c. X exceeds Z and either A or B must be less than 3.4.

3. Write a program to evaluate the constant e by the series

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

   Terminate the series when the next term would be less than 0.00000001 or when the number of terms exceeds 300. Print the series sum after each 20 terms, as well as the final value.

4. Write a program to read positive or negative data values; separate them into integer and fractional parts, and print a table of the numbers and their separate parts.

5. Write a program to read five data values from cards, print them in the order read, sort them into algebraically ascending order, and print them again in that order.

6. Write a program to compute a square root via an iterative process which uses the formula

   NEXTESTIMATE = (LASTESTIMATE + X/LASTESTIMATE)/2

   to obtain successive estimates for the square root of X. Use X/2 as the initial value for LASTESTIMATE, and terminate the iterative calculation when ABS(NEXTESTIMATE/LASTESTIMATE-1) < 0.00000001. Use the X values 0.0, 0.1, 0.2, 0.3, ..., 4.0, and print X, the common function SQRT(X), and the computed estimate of the square root.

CHAPTER 5 - FOR STATEMENTS

5.1 THE CONCEPT

In the solution of problems using numerical techniques, it is often desirable to execute a statement or series of statements in repetitive fashion. Suppose, for example, a program is to READ a value from a card, execute an assignment statement, and WRITE the result on the printer and this is to be performed for ten different input values. One way to do this would be to repeat the necessary statements ten times (in serial fashion) in the program. A better way, however, would be to write the necessary statements once and execute these same statements ten times.

In ALGOL this repetition or "looping" of statements is conveniently performed by means of a for statement. This statement permits the programmer to control the repetitive process for any number of statements. All statements which are to be repeated are under control of a variable called the controlled variable which is changed in value once per repetition, until some limit or condition is met as specified by the programmer. The following specifications completely describe the for statement:

⟨for statement⟩ ::= ⟨for clause⟩⟨statement⟩ | ⟨label⟩:⟨for statement⟩

⟨for clause⟩ ::= FOR ⟨controlled variable⟩←⟨for list⟩ DO

⟨controlled variable⟩ ::= ⟨simple variable⟩

⟨for list⟩ ::= ⟨for list element⟩ | ⟨for list⟩,⟨for list element⟩

⟨for list element⟩ ::= ⟨arithmetic expression⟩ |
⟨arithmetic expression⟩STEP
⟨arithmetic expression⟩UNTIL
⟨arithmetic expression⟩

The above definitions point out the use of four more reserved words, namely, FOR, DO, STEP, and UNTIL. It is also seen that the basic constituent of the for statement is the for clause. The for clause controls the iterative process for the statement following DO. The controlled variable takes on only one form—that of a simple variable and appears immediately after the reserved word FOR. The for list is the element that provides the initial and limiting condition to be applied to the statement following DO.

From the specifications given for a for statement it is seen that following the reserved word DO is a statement. In Chapter 3, we saw that a statement has many definitions, such as block, conditional statement, compound statement, for statement, etc. It should be kept in mind that any of these constructs may follow DO. Probably the most widely used statement following DO is the compound statement. This construct permits the programmer great flexibility in writing his programs.

The two types of for lists presented in the specifications above describe slightly different processes for the forming of loops in a program and therefore will be discussed separately. Before these are discussed, however, the student's attention is called to the recursiveness of the definition of the for list. Either, or both forms of the for list element may be repeated in the for list as long as they are separated by commas.

The process described by more than one for list element is exactly the same as if the for list elements were used to form a series of for statements. Each for list element would be taken in turn from left to right to form individual for statements. These for statements would be written using the same controlled variable and the same statement following the reserved DO as contained in the original for statement.

5.2 THE FOR LIST

The simplest for statement that can be written is when the for list element is a single arithmetic expression. This can be illustrated as follows (where AE represents any arithmetic expression):

```
FOR Y ← AE DO S1;S2
```

In this case, the value of AE is assigned to the variable Y and since there is no limiting condition, no test is made and the statement following DO (represented by S1) is executed. The for list is then considered exhausted and control will continue in sequence with the execution of the statement, represented by S2. Thus, in the above case, the same results are obtained by the series of statements:

Y ← AE; S1;S2

A more complete and useful extension of the above form is when the for list definition is used recursively as follows:

FOR J ← 3,4,A,C/D,SQRT(X) DO SR←SR + J/2;S2;

In this example, the for list consists of the arithmetic expressions 3,4,A,C/D,SQRT(X). The first value assigned to the controlled variable J, is a 3; the statement SR ← SR + J/2 is then executed. Next the value of 4 is assigned to J and the same statement is again executed. This process is repeated for all elements in the for list. After the last element (in our case, this is the value of SQRT (X)) has been assigned to J and the statement following DO has been executed, control will continue in sequence to the statement following the for statement (represented here by S2).

As the final example of this form of the for list element, assume that five values are to be read (one per card), their square roots taken, and the resultant values printed. The for statement to accomplish this is given as follows (note that in this case, variable IM is used chiefly as a counter):

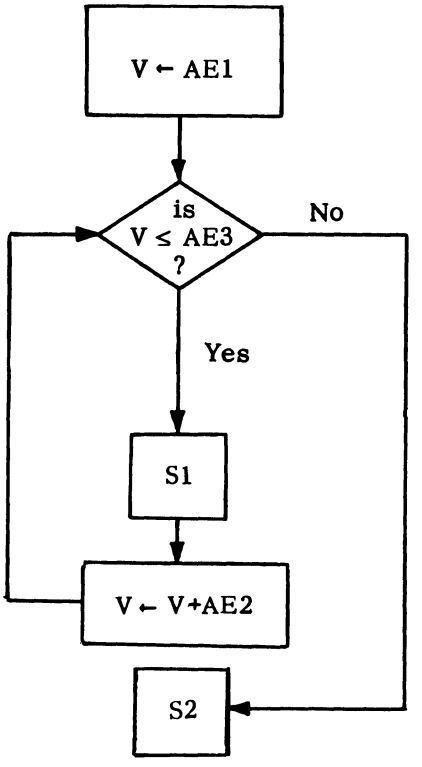
```
        FOR IM ← 1,2,3,4,5 DO
BEGIN
        READ(CARD,FIN,NUMBER);
        SQ ← SQRT(NUMBER);
        WRITE(LINE,FT2,IM,SQ,NUMBER)
END
```

In this example, the three statements enclosed between the words BEGIN and END make up a compound statement. This statement is then under control of the for clause and will be executed a total of five times.

The second form of the for list element is the STEP-UNTIL construct. Its general form is as follows:

FOR V ← AE1 STEP AE2 UNTIL AE3 DO S1; S2

In this case the initial value assigned to V is the value represented by AE1. If this value has not passed AE3 the statement following DO (represented by S1) is executed. All subsequent assignments to the controlled variable are equivalent to V ← V+AE 2 and are made immediately after the DO statement is executed. In all cases, before the DO statement is executed, a check is made to ascertain that the value of V has not passed AE3. The basic flow of the STEP-UNTIL construct can be illustrated by the following block diagram (in which AE2 is assumed to be positive):



AE1, AE2, and AE3 represent arithmetic expressions and may take on positive or negative values. This means that the for statement may either "step-up" or "step-down" during the repetitive process. The above block diagram shows that the DO statement may be executed zero or more times depending on the values of V and AE3. It also indicates that, if the value of AE2 is always zero, the for statement will be caught in an infinite loop. A complete description (considering all sign possibilities) of the STEP-UNTIL for list element follows:

```
            V ← AE1;

L2:         IF  AE2  = 0 OR (SIGN(AE2)  =  +1 AND V ≤ AE3) OR

            (SIGN(AE2) = -1 AND V ≥ AE3) THEN

BEGIN

            S1;
            V ← V + AE2;

            GO TO L2

END         ;
            S2
```

In the following example, if I = 1 and M = 0 the statement represented by S1 will not be executed at all; but rather control will transfer immediately to the statement represented by S2. If I=1, J=1, and M=1, statement S1 will be executed once and only once before transferring control to statement S2. However, if J=0, then statement S1 will be continually executed.

```
FOR K ← I STEP J UNTIL M DO S1; S2
```

The following example illustrates the use of the STEP-UNTIL construct in the same situation as found in the previous example.

```
                FOR IM ← 1 STEP 1 UNTIL 5 DO
BEGIN
                READ(CARD,FIN,NUMBER);
                SQ ← SQRT(NUMBER);
                WRITE(LINE,FT2 IM,SQ,NUMBER)
END
```

The same example could be written using a conditional statement.

```
                IM ← 1;
LOOP:           READ(CARD,FIN,NUMBER);
                SQ ← SQRT(NUMBER);
                WRITE(LINE,FT2,IM,SQ,NUMBER);
                IM ← IM+1;
                IF IM ≤ 5 THEN GO TO LOOP
```

5.3 USES OF THE FOR STATEMENT

Suppose it is necessary to find the natural logarithm for a series of numbers starting from 2 and going thru 3.1 in increments of 0.01 and to print all the results. This could be accomplished with the following for statement:

```
        FOR N ← 2 STEP .01 UNTIL 3.1001 DO
BEGIN
        ANS ← LN(N);
        WRITE(LINE,FT2,N,ANS)
END
```

In the above example, the number 3.1001 was used as the limiting value to insure that N will reach a value of 3.1. The programmer should always exercise caution in using real arithmetic values in this type of for list. The reason for this was discussed in Section 4.2

If another for statement follows DO, then the for statements are said to be nested. In this case the for list of the inner for clause is repeated for each element or step of the for list contained in the outer for clause. For example:

```
FOR I ← 0 STEP 2 UNTIL 10 DO
    FOR J ← 1,2,4 STEP 1 UNTIL 7 DO
        TBK ← TBK+IxJ;
READ(CARD,.....)
```

The assignment statement TBK ← TBK+IxJ is executed for I equal to 0 and for J equal to 1,2,4,5,6, and 7; then I is assigned the value of 2 and the assignment statement is again executed for all values of J. This process is repeated until the outermost "for loop" is exhausted after which the READ statement would be executed.

There are three aspects of for statements which the programmer should be aware of. The first is that a go to statement appearing outside the for statement may not transfer control to a labelled statement within the scope of a for statement. As illustrated:

```
        GO TO DIAG;
          .
          .
          .
          .
        FOR JX ← 1 STEP -2 UNTIL -11 DO
DIAG:   COMP ← COMP + 4/JX
```

In this case, the GO TO DIAG statement is not permitted as the controlled variable JX has not been assigned a value at the time of that particular entry. Control may be transferred to the for statement however, as follows:

```
          GO TO DIAG;
               .
               .
               .
               .
DIAG:     FOR JX ← 1 STEP -2 UNTIL -11 DO
          COMP ← COMP + 4/JX
```

The second condition to be aware of is the value of the controlled variable upon exit from the for statement. If a compound statement following DO contains a go to statement leading outside the for statement, the value of the controlled variable upon exit from the for statement will be the same as it was immediately preceding the execution of the go to statement. For example:

```
          TOT ← 0;
          FOR I ← 1 STEP 2 UNTIL 10 DO
BEGIN
          TOT← I+TOT;
          IF TOT > 5 THEN GO TO EXCEED
END;
EXCEED:   J←I
```

When control is transferred to label EXCEED the value of I will be 5 and any reference to I will use this value. If an exit from the FOR statement is due to exhaustion of the for list, then the value of the controlled variable is not defined.

Any statement following the for statement may access the variable I but the programmer cannot be assured of its value, unless it has been assigned some particular value after execution of the for statement.

Finally, it should be noted that the controlled variable has the status of any simple variable within the for statement. In other words, it may appear in arithmetic expressions or on the left side of an assignment statement.

Compute and print y, where

$$y = x \tan \quad +2 \ln \left| \cos \frac{x}{2} \right| ,$$

for values of x equal to 0.5, 0.5+n, 0.5+2n, etc. Terminate the computation when the next value of x will be greater than 1.5.

```
          BEGIN                                                          EVAL  1
FILE IN         CARD (2, 10);                                            EVAL  2
FILE OUT        LINE 1 (2, 15);                                          EVAL  3
FORMAT IN       FIN (F12.6);                                             EVAL  4
FORMAT OUT      FT1 (X10, "FORMULA EVALUATION - H. MAR"//);              EVAL  5
FORMAT OUT      FT2 (6(X4,F12.6,X4)/);                                   EVAL  6
REAL            X, N, Y, T;                                              EVAL  7
                WRITE (LINE, FT1);                                       EVAL  8
                READ (CARD, FIN, N);                                     EVAL  9
                WRITE (LINE, FT2, N);                                    EVAL 10
                FOR X ← 0.5 STEP N UNTIL 1.5 DO                          EVAL 11
          BEGIN                                                          EVAL 12
                T ← COS(X/2);                                            EVAL 13
                Y ← X × (SIN(X/2)/T) + 2 × LN(ABS(T));                   EVAL 14
                WRITE (LINE, FT2, X, Y)                                  EVAL 15
          END                                                            EVAL 16
          END  OF PROGRAM.                                               EVAL 17
```

Notice that the value of n is read in as input data and used as the increment in the for clause. The entire calculation is done in the single for statement with the initial value of x=0.5, assigned the first time through the statement. Subsequently x is incremented by n and tested against the limiting value 1.5.

```
          BEGIN                                                                  DHT1  1
FILE OUT        LINE 1 (2, 15);                                                  DHT1  2
FORMAT          FT1 (X10,"COMPARING VALUES OF SIN(X) GOTTEN TWO WAYS");          DHT1  3
FORMAT          FT2 (6(X4,F12.6,X4)/);                                           DHT1  4
COMMENT         A USE OF SIMPLE "FOR LIST ELEMENTS" TO GET SIN(X)                DHT1  5
                BY A POLYNOMIAL APPROXIMATION, EVALUATION OF                     DHT1  6
                (A×T) + (B×T↑3) + (C×T↑5) + (D×T↑7)                              DHT1  7
                IS DONE IN THE CONVENIENT NESTED FORM:                           DHT1  8
                T×(A + (T↑2)×(B + (T↑2)×(C + (T↑2)×D))),                        DHT1  9
                WHERE T = (2×X)/PI AND                                           DHT1 10
                A = 1.5708, B = -.6459, C = .0795, D = -.004 (ROUGHLY);          DHT1 11
REAL            X, POLLY, KOE, TSQ, T;                                           DHT1 12
                WRITE (LINE, FT1);                                               DHT1 13
                FOR X ← 0.070745 STEP .1 UNTIL 1.58 DO                           DHT1 14
          BEGIN                                                                  DHT1 15
                POLLY ← -.00436248; T ← (X+X)/3.14159; TSQ ← T×T;                DHT1 16
                FOR KOE ← .07948766, -.64592098, 1.57079485 DO                   DHT1 17
                POLLY ← KOE + TSQ × POLLY;                                       DHT1 18
                POLLY ← POLLY × T;                                               DHT1 19
                WRITE (LINE, FT2, X, POLLY, SIN(X))                              DHT1 20
          END   OF X LOOP;                                                       DHT1 21
          END                                                                    DHT1 22
```

EXERCISES

1. For what values of Y will the statement following DO be executed?

   FOR Y ← 3,4,6,8 STEP 3 UNTIL 18, 19, 23 DO

2. Evaluate the formula

   $h = Ax^2 + bx + c$

   for each of the values

   x = 0.12, -0.134, 0.5, -1.4, and 2.13

   where

   A = 2.4, B = 6.1, and C = 9.34

3. Write the following FOR clauses in the STEP...UNTIL form:

   a. FOR A ← 1,2,4,8,16,32,64,128,256 DO

   b. FOR B ← 1,0.5,0.25,0.125,0.0625 DO

   c. FOR C ← 0,0.5,0.75,0.875,0.9375 DO

   d. FOR D ← 100,90,81,72.9,65.61 DO

4. Sum the squares of the integers 1 thru n.

5. Repeat Exercise 3 of Chapter 4, utilizing the for statement whereever possible.

6. Find and print all of the prime numbers (which are necessarily odd numbers) greater than 3 and less than 1000. Recall that a prime number is an integer which is not factorable into integers other than 1 and the number itself. The only conceivable factors are the odd integers from N down to 3, for a candidate number N. If division by one of these yields an integer, the number N is not prime. (Do not overlook the possibility that SQRT(N) may yield values such as 8.9999999999 or 13.000000001 when N is a perfect square.)

7. Use a double FOR clause to evaluate the series:

$$\sum_{i}^{10} + \sum_{i}^{10} i^2 + \sum_{i}^{10} i^3 + \sum_{i}^{10} i^4 + \ldots + \sum_{i}^{10} i^8$$

(where $\sum^{n} i^m = 1^m + 2^m + 3^m + \ldots + n^m$.)

Also evaluate it in the form:

$$\sum_{1}^{8} 1^i + \sum_{2}^{8} 2^i + \sum_{3}^{8} 3^i + \ldots + \sum_{10}^{8} 10^i$$

(where $\sum^{n} m^i = m^1 + m^2 + m^3 + \ldots + m^n$)

8. Write a program which evaluates the series

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \text{----}$$

to eleven terms by means of a FOR list and the nested form of the polynomial. (Hint: 9! = 10!/10, 8! = 9!/9 etc.) Use x values from 0 to 10.00000 in steps of (x+1)/2. Compare, via printed output, these results with EXP(x).

CHAPTER 6 - SUBSCRIPTED VARIABLES

6.1 THE CONCEPT

The correct use of a subscripted variable is very important in the writing of ALGOL programs. It is imperative that the student thoroughly understand this concept. Therefore, the following pages will be devoted to some background for and usage of subscripted variables. The discussion will be based on a description of what is meant by a "set." With this concept firmly implanted, there should be no difficulty for the student in understanding the meaning and usage of a subscripted variable in ALGOL.

Mathematics defines a set by using such words as "class," "collection," "group," "family," etc. Additional phrases also help to define a set. For example:

a set of dishes
a set of books
a set of students at UCLA
a set of numbers
a set of pictures
a set of numbered cards
a set of houses in Denver

It is seen that each set or group represents many constituents of similar items. The constituents of a set are referred to as elements or members of that set. There may be five numbers in a set of numbers, twenty books in a set of books, or a thousand students in a set of students from UCLA. Each of the five numbers is an element of the set of five numbers, each book is an element of the set of twenty books, and each student is an element of the set of students. Each set is given a name; the elements or members of the set are commonly indicated by means of a subscript number.

For example, a set of four elements might be written as D_1, D_2, D_3, and D_4. These are read as "D sub 1," "D sub 2," "D sub 3," and "D sub 4." The numbers 1, 2, 3, and 4 are called subscripts and have no other function except to identify or label a particular element; much as house numbers are used to identify a set of houses or airplane numbers are used to identify each airplane of a particular airline.

To further exemplify this idea, let's take the example of a set of students from UCLA. Suppose there are five students with the names of Sam, Jane, Pete, Joe, and Bette. The set of these five students will be given the name CLICK; then $CLICK_1$ could represent Sam, $CLICK_2$ could represent Jane, $CLICK_3$ could represent Pete, etc. Thus, we see that five students, presumably each having something in common, all belong to the same set and each is a member or element of the set, identified by a subscript number.

As one more example, we take a set of numbered cards. Suppose that six cards are numbered as follows: 500, 501, 502, 503, 504, and 505. If the name of this set of numbered cards is DATA, then $DATA_1$ represents a card with the number 500 on it, $DATA_2$ represents a card with the number 501 on it, etc. In fact, we can go one step further and say that $DATA_1$ represents the number 500, $DATA_2$ represents the number 501, $DATA_3$ represents the number 502, etc.

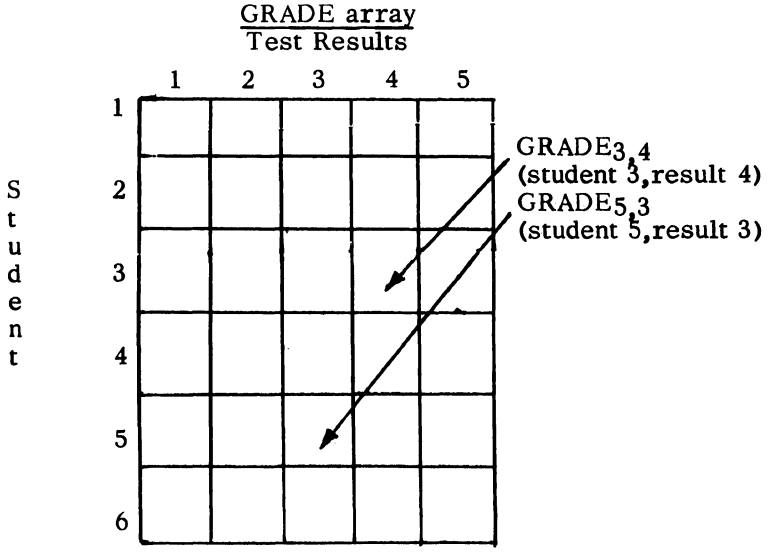
With this last example we can go into the concept of an array. In ALGOL, an array has a meaning very similar to that of a set in mathematics. The name of an array is called an array identifier; the subscript of an array element is called a subscript list. As in sets, an array represents a group or series of items—usually numbers. The elements of an array generally have something in common, either in meaning or usage. Suppose we have an array called GRADES where each element of this array represents the score of a different student for some particular test. For example, $GRADE_1$ represents the score of student 1 in the last history examination, $GRADE_2$ represents the score of student 2, $GRADE_3$ represents the score of student 3, etc. Hence, all the scores are referenced by one common name, and each score uniquely defined by a subscript number.

Now in another illustration, suppose there are 400 employees of company PQ. Each employee is donating part of his salary to a local charity. The amount that each employee contributes can be represented by the array identifier DONAT. Then $DONAT_1$ represents the contribution of employee number 1, $DONAT_2$ represents the contribution of employee number 2, and so forth. Assume also, that PAY is the identifier of an array which represents the salary of all employees, namely, PAY_1 is the salary of employee 1, PAY_2 the salary of employee 2, PAY_3 the salary of employee 3, etc., and suppose further, that the contribution of each employee is to be deducted from his pay check, then PAY_1 minus $DONAT_1$ computes the net pay of employee number 1, PAY_2 minus $DONAT_2$ will compute the net pay of employee 2, and so forth. Therefore, the net pay of all employees can be computed by using the same formula but varying the subscript numbers.

In the preceding example, the donation of each employee could have been represented by unique names. Thus, DONAT1 (this is not the same as $DONAT_1$) could have been used to represent the contribution of employee number 1, DONAT2 the contribution of employee 2, etc.; but this would require 400 different names to represent each employee. The concept of arrays, however, allows the use of one identifier to represent all employee donations and a subscript expression to reference each employee separately. Since, however, all notation in ALGOL is linearized (i.e., all symbols are written on the same line), it is necessary to use a slightly different notation for a subscripted variable than has been used in the above examples. Thus, in ALGOL, DONAT[1] is used to represent $DONAT_1$, DONAT[2] is used to represent $DONAT_2$, etc. The exact specifications for the writing of a subscripted variable will be given later.

The above discussion has been limited to arrays with a subscript list containing a single number or subscript expression. This type of array is referred to as a one-dimensional or linear array. If two subscripts are needed to identify elements of an array, then we say that this array is two-dimensional.

An example of a two-dimensional array is as follows: Assume six students are each taking five examinations. The results of these examinations can be placed in an array of 30 elements. We could use a one-dimensional array, but a two-dimensional array will enable the subscript numbers to have more meaning. For example, $GRADE_{1,1}$ represents the results of test 1 for student 1; $GRADE_{1,2}$ represents the result of test 2 for student 1 ..... and $GRADE_{1,5}$ represents the result of test 5 for student 1; $GRADE_{2,1}$ references the result of test 1 for student 2 ... and $GRADE_{5,5}$ references the result of test 5 for student 5. This array can be diagrammatically pictured as follows:



Here again the subscripts merely label what is being referenced. Picture that each square is a door with its appropriate subscript list on it, such as 1, 1 or 3, 4 or 5, 5, etc. These numbers identify each door; behind each door is the actual element we reference. The element may change its value, but the subscript list on the door will never change its value.

The exact specifications for writing a subscripted variable in an ALGOL program is given as follows:

⟨subscripted variable⟩ ::= ⟨array identifier⟩[⟨subscript list⟩]

⟨array identifier⟩ ::= ⟨identifier⟩

⟨subscript list⟩ ::= ⟨subscript expression⟩|⟨subscript list⟩,
⟨subscript expression⟩

⟨subscript expression⟩ ::= ⟨arithmetic expression⟩

The subscript list specifies the element number within the array and is separated from its array identifier by means of the bracket symbols] and [If there is only one subscript expression in a subscript list then the array is said to be a single-dimensioned array, otherwise it is called a multi-dimensioned array. The subscript expressions of a multi-dimensioned array are separated from each other by means of a comma. There is no limit to the number of dimensions an array can have; for practical purposes, however, more than three or four dimensions are rarely used. Note that the subscript expression is defined to be an arithmetic expression. This means that all the rules given in Chapter 2 for the arithmetic expression can be applied to the subscript expression. In fact, the subscript expression can itself contain another subscripted variable.

Each subscript expression is evaluated to yield a single numeric value. This value is treated as though it were of type INTEGER. If the evaluation of a subscript expression yields a value of type REAL or if in fact the subscript expression is a REAL number, then the following operations are automatically invoked before the value is used to obtain an element of an array:

Subscript value = ENTIER (value of subscript expression + 0.5).

Each subscript expression may take on positive or negative values; but the range of each subscript expression may not exceed 1023 elements. Thus, a subscript expression could take on values from 0 thru 1022 or -10 thru 1012 or 5000 thru 6022.

Verify the following expressions containing subscripted variables:

MAT[-4]

A+D-MAT[I]

MAT1[JK]

```
STUB[3,I]

MIX[J]-MIX[I]+2.0

CAD[A+SQRT(C)/2, B-N*2]

INV[M+N/3.4]

SORT(A[I]+INV[2]/3)+JOB

CHECK[TEST[J]]

CHECK[TEST[TEST[I]]]

MAN[I,J,K,L[K]+2]

A[I]+1
```

6.2 ARRAY DECLARATION

The information that a given variable is subscripted is conveyed by means of the array declaration. This declaration also contains information as to the type of the array (namely, REAL or INTEGER), the number of dimensions in a given array, and the upper and lower bounds of each dimension. The syntax for the array declaration follows:

⟨array declaration⟩ ::= ⟨array kind⟩ ARRAY ⟨array list⟩

⟨array kind⟩ ::= ⟨empty⟩ | ⟨type⟩

⟨array list⟩ ::= ⟨array segment⟩ | ⟨array list⟩,⟨array segment⟩

⟨array segment⟩ ::= ⟨array identifier⟩ [⟨bound pair list⟩] |
⟨array identifier⟩,⟨array segment⟩

The above specifies that the array list is basically made up of an array identifier followed by [⟨bound pair list⟩]. If the array kind is empty then the type of the array is assumed to be REAL. The bound pair list is defined as follows:

⟨bound pair list⟩ ::= ⟨bound pair⟩ | ⟨bound pair list⟩, ⟨bound pair⟩

⟨bound pair⟩ ::= ⟨lower bound⟩:⟨upper bound⟩

⟨lower bound⟩ ::= ⟨integer⟩

⟨upper bound⟩ ::= ⟨integer⟩

Note that both the upper bound and lower bound of each dimension of the array must be specified and that they are separated from each other by means of the colon. If the program accesses any element outside this range an error message is given and the program is terminated. If more than one array has the same bound pair list and the arrays are all of the same type, the bound pair list need not be repeated for each array but rather follows the right-most array identifier. The number of elements in any one dimension of an array may not be greater than 1023.

The following are legal ways to write an array declaration:

```
ARRAY MAT[0:240], MAT1 [0:240];

REAL ARRAY STUB[0:4,1:6],CAD[ 400:500,1:50 ];

INTEGER ARRAY CHECK[-60:-4],MAN[0:5,0:25,1:7,4:15];
```

or

```
ARRAY  MAT,MAT1[0:240],  STUB[0:4,1:6],  CAD[400:500,1:50];

INTEGER ARRAY CHECK [-60:-4];

INTEGER ARRAY MAN [0:5,0:25,1:7,4:16];
```

These declarations specify, for example, that array MAT contains 241 elements; the value of the subscript expression may vary from 0 thru 240, any other value would constitute an error.

It should be mentioned at this point that the most efficient lower bound of any bound pair list is zero. This efficiency is dictated by the B 5500 hardware. Although the extra time associated with a non-zero lower bound is insignificant, the student should realize that larger programs written by efficiency-minded programmers will most generally declare arrays with a lower bound of zero—whether the zeroeth element is used or not.

Wherever possible a multi-dimensioned array should be arranged so that the right-most subscript expression of the subscript list will be the one which is changed the most rapidly.

6.3 USE WITH THE FOR STATEMENT

One of the most frequent uses of the subscripted variable is with the for statement. In this case, it is possible to repeatedly execute a statement where the same subscripted variable accesses a different value each time the statement is executed. This is accomplished by using the controlled variable in the subscript list of the subscripted variable. Whenever the controlled variable changes its value, a different element of the array is accessed. Suppose that an array is to be searched for the first element containing a value of zero. Then the following for statement will transfer control to a statement with the label ZERO if such an element is found:

```
FOR I ← 1 STEP 1 UNTIL 500 DO

IF MAT [I] = 0 THEN GO TO ZERO;
```

Another illustration would be when a single for statement is used to figure the net pay of X-numbered men. In this example, each man is represented by a subscript number. Arrays are set up to contain the values of each man's pay rate, his social security tax, donations, etc. Another array is established to represent his net pay. Only one for statement is required to compute the net pay for all the men by varying the controlled variable so that each reference to an array accesses information for that man only, as follows:

```
FOR I ← 1 STEP 1 UNTIL X DO

MAN[I] ← PAY[I] - FICA[I] - TAX[I] - DONATION[I];
```

The last example will illustrate the "zeroing" out of the two-dimensional array DATA. Assume the array is declared as ARRAY DATA [0:10, 1:12] and that all elements are to be given the value of zero.

```
FOR I ← 0 STEP 1 UNTIL 10 DO

FOR J ← 1 STEP 1 UNTIL 12 DO

DATA [I, J] ← 0
```

An illustration of the use of subscripted variables is given in the following example. This simplified program will compute and print the weekly net pay for 100 employees.

Three cards are read for each employee. Card 1 contains the employee number, card 2 the gross pay, and card 3 the number of employee dependents.

```
          BEGIN                                                          PAY   1
FILE IN         CARD (2, 10);                                            PAY   2
FILE OUT        LINE 1 (2, 15);                                          PAY   3
FORMAT IN       FIN (F12.6);                                             PAY   4
FORMAT OUT      FT1 (X10, "SIMPLIFIED PAYROLL - A. RASCAL"//);           PAY   5
FORMAT OUT      FT2 (6(X4, F12.6, X4)/);                                 PAY   6
ARRAY           EMPNO, GRSS, DEPEND, NET[0:99];                          PAY   7
INTEGER         I;                                                       PAY   8
REAL            TAX;                                                     PAY   9
                FOR I ← 0 STEP 1 UNTIL 99 DO                             PAY  10
                READ (CARD, FIN, EMPNO[I], GRSS[I], DEPEND[I]);          PAY  11
                FOR I ← 0 STEP 1 UNTIL 99 DO                             PAY  12
          BEGIN                                                          PAY  13
                TAX ← (GRSS[I] - 13 × DEPEND[I]) × 0.14;                 PAY  14
                NET[I] ← GRSS[I] - TAX                                   PAY  15
          END   ;                                                        PAY  16
                WRITE (LINE, FT1);                                       PAY  17
                FOR I ← 0 STEP 1 UNTIL 99 DO                             PAY  18
                WRITE (LINE, FT2, EMPNO[I], GRSS[I], NET[I])             PAY  19
          END   .                                                        PAY  20
```

EXERCISES

1. Write a for statement with 3 for clauses to store 1.0 in all elements of array Z declared as follows:

   Z[0:5,0:-6,-10:10]

2. Array A contains 50 numbers A[0:49]. Write a for statement to check each element for <0 and keep a count of all such numbers in variable CT.

3. Fill the elements of a 10 by 6 array, A, with values such that:

   when $I < J$, $A[I,J] = I/J$

   when $I = J$, $A[I,J] = 0$

   when $I > J$, $A[I,J] = J/I$

   (where $0 < I < 11$ and $0 < J < 7$).

   Print the array, six elements to a line by means of a write list of the form A[I,1], A[I, 2], A[I,3], ..., A[I,6].

4. Generate a 10 by 10 array similar to that of exercise 3. Then compute and print the summations for row I and column I for I values from 1 to 10.

5. Re-do exercise 5 of Chapter 4, using up to 60 data values.

6. Re-work the illustrative program of Chapter 3, using a 2 by 3 array to store the incoming data. Perform the computations in a for statement, using subscripted variables. To make it more interesting, the problem might be done for 3 equations in 3 unknowns.

7. Re-work the illustrative program of Chapter 5 where the sine function is computed via a nested polynomial. Store the polynomial coefficients in a one-dimensioned array and write the nested polynomial using a subscripted variable for the coefficients.

CHAPTER 7 - COMMUNICATION - DATA AND RESULTS

7.1 FILE DECLARATION

In previous chapters, the reading of input data and the printing of the calculated results have been accomplished through the use of a read or write statement using identifiers which were defined by a fixed set of declarations placed at the beginning of the program. Consider the following examples:

READ (CARD, FIN, ⟨list⟩) [⟨label⟩]

WRITE (LINE, FT2, ⟨list⟩)

In each of the above examples, the first element within the parentheses is an identifier, the declaration of which appeared as one of the fixed set at the beginning of the program. The declarations which defined CARD and LINE were:

FILE IN CARD (2,10)

FILE OUT LINE 1 (2,15)

The purpose of these declarations, called file declarations, is to associate an identifier with a particular data file and a specific method for handling the file. The above declarations associate the identifiers CARD and LINE with given files. This allows the programmer to call data files by symbolic names without regard to physical equipment requirements, such as, which card reader, etc. How the equipment is actually assigned will be discussed later. The identifiers may be chosen arbitrarily within the normal restrictions on identifiers.

For purposes of this text a file will be considered as either an input file consisting of punched cards or an output file consisting of information to be printed on the line printer. A single file, which will be uniquely distinguished by its identifier, will be a particular, logical grouping of information. For example, all of the input data for a program may be considered as a file.

The syntactical definition of the file declaration follows. The complexity of the definition cannot be avoided since card input and printer output are a part of the language facilities designed to handle all kinds of input-output media.

⟨file declaration⟩ : : = FILE ⟨I-O part⟩⟨file part⟩

⟨I-O part⟩ : : = IN | OUT

⟨file part⟩ : := ⟨file identifier⟩⟨unit designation⟩(⟨buffer part⟩)

⟨file identifier⟩ : : = ⟨identifier⟩

⟨unit designation⟩ : := ⟨empty⟩ | ⟨space⟩1

⟨space⟩ : : = ⟨single space⟩ | ⟨space⟩⟨single space⟩

⟨buffer part⟩ : := ⟨number of buffers⟩, ⟨buffer size⟩

⟨number of buffers⟩ : : = ⟨unsigned integer⟩

⟨buffer size⟩ : : = ⟨unsigned integer⟩

From the syntax, it is shown that the file declaration begins with the reserved words FILE IN or FILE OUT. This is followed by the file part.

The file part consists of a file identifier followed by a unit designation followed by a buffer part in parentheses. The file identifier may be any valid identifier and can often be chosen with mnemonic appeal. For example, CARD implying card input data.

The unit designation must be empty for card input files. For printer files, the digit one, separated from the file identifier by at least one blank, must be used. This is the only exception to the discussion on blanks in Section 2.3.

The buffer part gives the number and size of the buffers in terms of unsigned integers. These buffers are arbitrary areas in memory which provide temporary storage for character information that is received from the card reader or transferred to the printer. The use of a buffer will be discussed further in the next section. The only consideration that the programmer must give to these buffers is their number and size. The number of buffers should be chosen according to the amount of data being read or written at a given time. Often multiple buffers will cause a given program to run more rapidly. For many problems, however, two buffers will be nearly an optimum choice.

The information contained in one punched card is interpreted as 80 alphanumeric characters. This requires a buffer of ten words since each buffer word holds eight characters. Since we can print 120 alphanumeric characters on a line of output, a buffer of 15 words is required for one line of print. This explains the occurrence of 10 and 15 in the buffer size part of our previous examples.

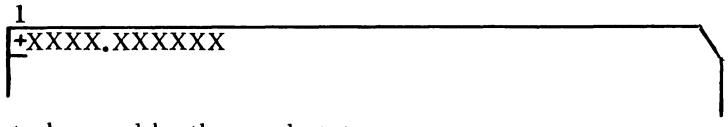
For many programs, the two file declarations previously used are adequate. The programmer might consider changing the buffer part to specify more than two buffers, if the program is highly dependent on input or output.

7.2 FORMAT DECLARATION

The second identifier which appears in the parenthetical expression associated with both the read statement and the write statement is a format identifier. This identifier refers to precise specifications for the editing of input or output data. This identifier is associated with its corresponding editing specifications in a format declaration.

It is necessary to specify this editing information, since data in an external representation is in the form of characters which must be converted into numerical values in the computer. (Care should be taken when comparing data read from cards with program constants since the converted input value may not be exact in the least significant digit position; i.e., @+12 may not exactly equal 0.00000001@+20 where the former is a constant and the latter an input value.)The programmer must indicate in what form the external data is furnished, in the case of input, or is to appear, in the case of output. For card input the programmer must specify exactly how each column of the card is to be treated when read. Similarly, it must be indicated exactly how each column of a printed line is to appear. It is usually best to lay out the desired format, possibly on grid paper, to facilitate the writing of editing specifications to obtain this format.

The input capability used up to now allowed a card of the form



to be read by the read statement

READ (CARD, FIN, ⟨list⟩)

The identifier FIN refers to the format for editing this data as it is read into the computer. This format, recall, was of the form F12.6. This says to treat only the first twelve columns on the card, where there are six digits to the right of the decimal point (which

is in column 6), as a string of characters to be translated into the machine representation of a real number.

On input, the information from a card is deposited in the buffer in character form. It is then converted, according to the editing specifications, into the internal numerical form. In the output case, the program deposits the character information, according to output editing specifications, into the buffer. From there, it is automatically transferred directly to the printer.

For reference purposes, in the following discussion, it will be valuable to list the syntax of the format declaration.

⟨format declaration⟩ ::= FORMAT⟨I-O part⟩⟨format part⟩ |
FORMAT ⟨format part⟩

⟨I-O part⟩ ::= IN | OUT

⟨format part⟩ ::= ⟨format identifier⟩ (⟨editing specifications⟩) |
⟨format part⟩,⟨format identifier⟩
(⟨editing specifications⟩)

⟨format identifier⟩ ::= ⟨identifier⟩

⟨editing specifications⟩ ::= ⟨editing segment⟩ |
⟨editing specifications⟩/ |
/⟨editing specifications⟩ |
⟨editing specifications⟩
⟨editing segment⟩

⟨editing segment⟩ ::= ⟨editing phrase⟩ |
⟨repeat part⟩(⟨editing specifications⟩) |
⟨editing segment⟩,⟨editing phrase⟩ |
⟨editing segment⟩,⟨repeat part⟩
(⟨editing specifications⟩)

⟨editing phrase⟩ ::= ⟨repeat part⟩⟨editing phrase type⟩

⟨field part⟩ | ⟨string⟩

⟨repeat part⟩ ::= ⟨empty⟩ | ⟨unsigned integer⟩

⟨editing phrase type⟩ ::= E | F | R | I | X

⟨field part⟩ ::= ⟨field width⟩ | ⟨field width⟩.⟨decimal places⟩

⟨field width⟩ ::= ⟨unsigned integer⟩

⟨decimal places⟩ ::= ⟨unsigned integer⟩

⟨string⟩ ::= "{any string of valid characters except"}"

The syntax indicates that the declaration begins with the reserved word FORMAT, which, may at the option of the programmer, be followed by IN or OUT. Next, there is an identifier which is to be associated with the adjacent editing specifications.

The editing specifications, themselves, are made up of editing segments which are in turn made up of editing phrases. It is the editing phrase type and its use in forming more complex structures which must now be considered. The editing phrase is the basic mechanism for describing the conversion of character strings into numeric form and vice versa. On output it also performs the additional function of transmitting a fixed string of characters, given as an editing phrase.

For the editing of input information, the F, E, and R type editing phrases treat numbers with decimal points, converting them into real numbers internally. The I type is applicable to integers. The X type phrase, when applied to input, indicates that a certain number of columns on the card are to be ignored.

When editing output, the E, F, R, and I phrases have an analogous function to the above, that is, they produce either decimal numbers or integers in the output string. The X phrase produces a specific number of blank spaces in the output line.

The number of characters to be treated as a group by a particular editing phrase and the position of the decimal point, in the case of E, F, and R type editing phrases, is given by the field part. The field width is an unsigned integer specifying the number of character positions to be edited. The decimal places are also indicated by an unsigned integer and specify the number of character positions between the decimal point and the right end of the field.

For example, F6.2 would say to treat the contents of a field six characters wide, where the fourth character will be a decimal point, as a decimal number. This would be used to edit a number such as -32.05 appearing on a card.

The following tables and associated discussion will clarify the use of the various editing phrase types for both input and output.

INPUT EDITING PHRASES

Editing Phrase Type	Editing Phrase Example	Processed As	Type of Variable Being Initialized	Example of Field Contents
E	E9.2	6-bit characters	REAL	+7.18@-03
F	F7.1	6-bit characters	REAL	-3892.5
R	R8.0	6-bit characters	REAL	-5.1362 63.0E10
I	I6	6-bit characters	INTEGER	-76329
X	X7	6-bit characters	None	any 7 characters

The definition of each input editing phrase type is given below:

E - Initializes a variable to the number found in the field described by field width. Field width must be at least 7 greater than decimal places, since the input data is required to be in one of the following forms:

```
-n.ddd---d@±ee
+n.ddd---d@±ee
 n.ddd---d@±ee
nn.ddd---d@±ee
```

A two-digit exponent must be in the rightmost two characters of the field. Preceding it, must be a sign, and preceding the sign is an @ symbol. Next, to the left, is a number of digits equal to decimal places in the editing phrase, preceded by a decimal point. To the left of the decimal, there may be either two digits or a sign and one digit. The sign may be indicated by +, -, or a single space which is interpreted as positive.

F - Initializes a variable to the number found in the field described by field width. The input data must be in one of the following forms:

```
+nn---n.dd---d
-nn---n.dd---d
 nn---n.dd---d
nnn---n.dd---d
      +.dd---d
      -.dd---d
       .dd---d
```

The field width must be at least two greater than the decimal places to accommodate the decimal point and a possible sign. A decimal point must be present; zero or more digits may precede it. Finally, there must be one or more digits after the decimal point; the number of these digits must equal decimal places in the editing phrase. The rightmost decimal place must fall in the rightmost character of the field. If the sign is omitted, the number is taken to be positive.

R - Initializes a variable to the number found in the field described by field width. It will treat data in either the E or the F formats with the following features:

1. The number can be positioned anywhere in the field.

2. The actual position of the decimal point in the number overrides that specified in the decimal place part of the editing phrase.

3. The & is accepted as a +.

4. An E may be used instead of @.

5. The exponent part need not contain a sign position.

I - Initializes a variable to the integer found in the field described by field width. The least significant digit must fall in the rightmost character of the field. The sign need only occur if it is negative.

OUTPUT EDITING PHRASES

Editing Phrase Type	Editing Phrase Example	Processed As	Type of Evaluated Expression	Example of Field Contents
E	E11.4	6-bit characters	REAL	-0.0125@+02
F	F9.3	6-bit characters	REAL	6735.125
R	R10.3	6-bit characters	REAL	-1328.001 -1.576@+05
I	I6	6-bit characters	INTEGER	346140
X	X8	6-bit characters	None	8 blanks

Each output editing phrase type is defined below.

E - Places the value of one expression in the field described by field width. This value has the following form when placed in the output data string:

n.dd---d@±ee
-n.dd---d@±ee

Again, the field width must be 7 greater than the decimal places. If this is violated, an asterisk is printed in place of the number. If field width is more than 7 greater than decimal places, leading single spaces are used to complete the field. The value of the expression is rounded to one more than the number of places indicated by decimal places of the editing phrase. The sign of the number, the first significant digit, and a decimal point occur first. If the number of significant digits in the expression value is less than decimal places plus one, trailing zeros are appended. To complete the field, the symbol @, the sign of the exponent, and the appropriate two-digit exponent are inserted.

The sign of the number is represented by a single space if positive, and a minus sign if negative. The sign of the exponent is either + or -.

F - Places the value of one expression in the field described by field width. This value has the following form when placed in the output string:

```
 nn---n.dd---d
-nn---n.dd---d
nnn---n.dd---d
```

The expression value is rounded to the number of designated decimal places and that number of decimal places will be printed. If the digits preceding the decimal point and the sign of the number (if negative) do not fill the remainder of the field, leading single spaces will be used to complete the field. If the specified field width is inadequate for printing the number, an asterisk will be printed instead of the number.

R - Places the value of one expression in the field described by the field width. Depending on the size of the number to be converted, a choice is made between the E and F formats. If the number will fit into the format described by the field part, F type editing is used. Otherwise, it is edited using E type editing.

I - Places the value of the expression in the field described by the field width. The expression value is rounded to an integer and placed completely to the right in the field, preceded by a minus sign for a negative number and leading single spaces, if any.

The sign of the number is the same as for the E editing phrase type.

Again, an asterisk is printed if the number of digits in the expression value exceeds field width for positive values or exceeds field width minus one for negative values.

X - Places a number of single spaces, as indicated by field width, in the output string.

Each editing phrase describes a portion of the input data being read or the output data being written. If the same editing phrase or series of editing phrases occur several times, the repeat part may be used to conserve writing. For example, the series of editing phrases F5.2, F5.2, F5.2 may be written 3F5.2. Similarly I3, X3, F5.2, I3, X3, F5.2 is equivalent to 2(I3, X3, F5.2). If the repeat part is omitted before an editing phrase or phrases enclosed in parentheses, the effect is the same as if the repeat part were infinite and a slash were to appear just inside the right parenthesis. The occurrence of a slash (/) within a series of editing specifications causes a skip to the next record. In card input, this causes the remainder of the present card to be ignored and the next one read if more data is required. In printer output, it causes the remainder of the line to be filled with blanks. Subsequent data will then appear on the following line or lines.

The series of editing phrases which occur in a particular set of editing specifications is applied to the data from left to right. When the end of a set of editing specifications is reached or the end of

the list part of the I-0 statement has been reached, the same action is effected as that which occurs when a slash is encountered. If, when the end of a set of editing specifications is reached, more items remain in the list, the format is reused from the beginning. If a slash follows the editing specifications for a complete record, the effect of two slashes is obtained.

Note that the syntax requires that commas be used to separate editing phrases within an editing segment. The quote marks at the ends of a string, which is one type of editing phrase, do not relieve the need for the comma. A comma may not appear adjacent to a slash since the comma is needed only within an editing segment.

7.3 LIST DECLARATION

The list declaration associates a set of arithmetic expressions with an identifier. Previously, in reading or writing, it was necessary to include the list in the read or write statement. The same effect is obtainable by the use of a list identifier in the I-0 statement. Formally the list declaration appears as follows:

⟨list declaration⟩ ::= LIST ⟨list part⟩

⟨list part⟩ ::= ⟨list identifier⟩(⟨list⟩)|⟨list part⟩, ⟨list identifier⟩(⟨list⟩)

⟨list identifier⟩ ::= ⟨identifier⟩

⟨list⟩ ::=⟨list segment⟩ | ⟨list⟩, ⟨list segment⟩

⟨list segment⟩ ::= ⟨expression part⟩ | ⟨for clause⟩⟨list segment⟩ | ⟨for clause⟩[⟨expression list⟩]

⟨expression part⟩ ::= ⟨arithmetic expression⟩

⟨expression list⟩ ::= ⟨expression part⟩ | ⟨expression list⟩, ⟨expression part⟩ | ⟨list segment⟩ | ⟨expression list⟩, ⟨list segment⟩

The syntax indicates that the list is a declaration containing a series of list identifiers and the list segments associated with them. Several examples should serve to illustrate the possible forms that the list may take.

LIST L1 (A, B, C, D)

This declaration associates the list of variables A, B, C, and D with the identifier L1. If it is used in a read statement it will cause four variables to be initialized with input data. In the output case, it will cause the values of the four variables to be printed.

```
LIST  AFT (X, Y+2, Z*2, SQRT(X+Y), 1.5)
```

The list AFT may only be used in a write statement since it contains expressions. These expressions will be evaluated, using the current values of the variables, before being printed. Since a constant is a valid arithmetic expression it may also appear as a list element for output. A minor restriction on the use of function designators in output list elements is discussed at the end of Section 10.4. If only variables appear as elements of the list, the list may be used for both reading and writing. L1 above is such a list.

```
LIST  INPT (A, B, FOR I ← 1 STEP 1 UNTIL 3 DO  X[I])
```

This list might be used in reading five input values. The first two will be assigned to A and B. The remaining three values will be assigned to the array elements X[1], X[2], and X[3]. In other words, a for clause followed by a subscripted variable may appear in a list for use in reading data into an array.

```
LIST TWOARRAY (FOR I ←1 STEP 1 UNTIL 3 DO  [A[I], B[I], C[I]])
```

This will cause the first three input data values encountered to be assigned to A[1], B[1], and C[1], the next three to A[2], B[2], and C[2], etc. It is possible to nest for clauses for use in reading data into arrays of more than a single dimension.

One list declaration may contain several lists separated by commas.

```
LIST  A1 (A, B, C),  LIN (X+Y, Z*2), OUPUT (FOR I ← 1 STEP 1
        UNTIL  3  DO  FOR  J ← 1  STEP  1 UNTIL N DO
        [Q[I,J], P[I,J]])
```

7.4 READ STATEMENT

The read statement actually causes values to be read from cards and assigned to specific variables. Recall that the read statement, which we have been using, actually refers to identifiers whose meanings have been spelled out by declarations. The syntax of the read statement follows:

⟨read statement⟩ ::= READ (⟨input parameters⟩) ⟨action label⟩

⟨input parameters⟩::= ⟨file identifier⟩,⟨format identifier⟩, ⟨list⟩ |
⟨file identifier⟩,⟨format identifier⟩,
⟨list identifier⟩

⟨action label⟩::= [⟨end of file label⟩] | ⟨empty⟩

⟨end of file label⟩::= ⟨label⟩

The file and format identifiers refer symbolically to the file from which the data is to be read and the specifications which are to be used in editing it. The list part may be an identifier referring to a list which is specified in a declaration, or it may be an actual list.

The action label is a label to which control is transferred when an attempt is made to read more data than exists. Obviously, it must be a label which has been appropriately declared.

When a read statement is executed, reading continues until all variables in the list have been assigned values. This requires careful attention to the one-to-one correspondence which must exist between the input data elements, the format editing information, and the list elements up to the end of the list. Recall that the input character string is initially brought into a buffer from the card reader. The editing specifications are used to extract data values from the buffer and convert them into internal numerical representation. These values are assigned sequentially to the variables in the list. This process can occur simultaneously with the filling of other buffers, if more than one has been specified. The refilling of a buffer is initiated automatically, as soon as its present contents have been edited.

7.5 WRITE STATEMENT

The write statement is highly analogous to the read statement. Its obvious function is to print out the values of elements which appear in its list. It can also be used to print strings of characters given in format declarations. Syntactically it appears below.

⟨write statement⟩ ::= WRITE (⟨output parameters⟩)

⟨output parameters⟩ ::= ⟨file identifier⟩
⟨format and list parameters⟩

⟨format and list parameters⟩ ::= ⟨format and list part⟩|⟨empty⟩|
[⟨skip to next page⟩]

⟨format and list part⟩ ::= ,⟨format identifier⟩|
,⟨format identifier⟩, ⟨list⟩|
,⟨format identifier⟩, ⟨list identifier⟩

⟨skip to next page⟩ ::= PAGE

The syntax points out several interesting features of the write statement. It can be used, in a form much like a read statement, to refer to a file, a format, and a list to print values of the expressions in the list.

Notice that, according to the syntactical definition, the file identifier must always be present while the format and list part might be empty. A write statement of the form

WRITE (F1)

where F1 is a file identifier is perfectly valid. It would cause the printer to skip one line.

If the format and list part consists of the skip to next page alternative,

WRITE (F1[PAGE])

the printer paper will be advanced to the top of the next page. The skip to next page alternative should not appear with a format and list part.

Headings and other constant information can be printed using the alternative definition of the format and list part which says that it may be a format identifier alone.

Notice that a comma must separate the file identifier from the format and list part. Similarly a list part must be separated from the format identifier by a comma. The list part is defined syntactically in the previous section.

There must be a correspondence between numeric values to be printed, editing specifications, and list elements, just as there was in the input process discussed in the previous section. The entire output process is analogous to that of input except that the direction of information flow is reversed.

It is possible to expand the example of Chapter 6 so that various I/O editing can be illustrated. Note that, in this example, a slightly different approach has been taken, in that no arrays are used.

Assume that the pertinent data for each employee is punched on an input card of the following format:

Cols. 1-5 - employee number (I5)

Cols. 9-14 - hourly base rate of pay (F6.2)

Cols. 18-20 - number of hours worked - to the nearest hour (I3)

Cols. 24-27 - overtime factor for hours over 40 (F4.1)

Cols. 31-33 - number of dependents (I3)

Cols. 37-42 - weekly donation to United Fund (F6.2)

Cols. 46-51 - weekly contribution for savings bonds (F6.2)

```
          BEGIN                                                                  TAX   1
FILE IN         CARD(2,10);                                                      TAX   2
FILE OUT        LINE 1 (2,15);                                                   TAX   3
FORMAT          FMI(I5,X3,F6.2,X3,I3,X3,F4.1,X3,I3,2(X3,F6.2) ) ;                TAX   4
FORMAT          FMTH("EMPLOYEE",X14,"HOURS    UNITED FUND",X5,"SAVINGS B"TAX   5
                     "OND     GROSS" /" NUMBER    BASE PAY   WORKED",X3, TAX   6
                     "CONTRIBUTION    CONTRIBUTION",X5,"PAY",             TAX   7
                     X7,"NET PAY",X7,"TAX" /);                           TAX   8
FORMAT          FMTD(X1,I5,X6,F6.2,X4,I4,X7,F6.2,X10,F6.2,X8,2(F7.2,X5),  TAX   9
                     F7.2);                                              TAX  10
FORMAT          FMTD(X10,"----------  ------    --------",X8,            TAX  11
                     "--------     ----------  ---------"),              TAX  12
                FMTT(" TOTAL =   ",F7.2,X2,I6,X5,F8.2,X8,F8.2,X5,F10.2,  TAX 13
                     X3,F9.2 );                                          TAX 14
REAL            BASE,NET,UF,BOND,GRS,NETT,BASET,UFT,BONDT,GRST,OVERT,TAX;TAX  15
INTEGER         EMP,HRW,HRWT,DEP;                                        TAX  16
LABEL           FINIS,MORE ;                                             TAX  17
COMMENT         PRINT HEADINGS;                                          TAX  18
                WRITE(LINE,FMTH);                                        TAX  19
COMMENT         READ CARDS AND PRINT RESULTS UNTIL END OF FILE REACHED;  TAX  20
MORE:           READ(CARD,FMI,EMP,BASE,HRW,OVERT,DEP,UF,BOND)[FINIS];    TAX  21
COMMENT         COMPUTE GROSS PAY FROM BASE PAY AND OVERTIME;            TAX  22
                IF HRW > 40 THEN                                         TAX  23
                GRS ← 40 × BASE + OVERT×BASE × (HRW-40) ELSE             TAX  24
                GRS ← HRW × BASE;                                        TAX  25
COMMENT         COMPUTE TAX ;                                            TAX  26
                TAX ← (GRS - 13 × DEP) × 0.14; IF TAX < 0 THEN TAX ← 0;  TAX 27
COMMENT         COMPUTE NET PAY AND PRINT;                               TAX  28
                NET ← GRS - TAX - BOND - UF ;                            TAX  29
                WRITE (LINE,FMTD,EMP,BASE,HRW,UF,BOND,GRS,NET,TAX);      TAX  30
COMMENT         CUMULATE TOTALS;                                         TAX  31
                BASET ← BASET + BASE;   HRWT ← HRWT + HRW;               TAX  32
                UFT ← UFT + UF ; BONDT ← BONDT + BOND;                   TAX  33
                GRST ← GRST + GRS;  NETT ← NETT + NET ;                  TAX  34
                GO TO MORE;                                              TAX  35
FINIS:          WRITE(LINE,FMTD);                                        TAX  36
COMMENT         PRINT TOTALS;                                            TAX  37
                WRITE(LINE,FMTT,BASET,HRWT,UFT,BONDT,GRST,NETT);         TAX  38
          END.                                                           TAX  39
```

EXERCISES

1. Write formats for the following output table with appropriate spacing between columns for readability.

Number	Area	Diameter
x	x.xxxx@±xx	xxxx.xx

2. Day, month, year, and page are variables containing two digit integers. What are the necessary format declarations and write statement(s) to print the heading

   TODAYS DATE IS xx/xx/xx. PAGE xx

3. Lay out a data card in the following format

   FORMAT FN(X3,I6,F6.2,E20.6,X10,R12.6).

4. Print a checkerboard, using the asterisk to form the lines and the letter X to fill in the dark squares. (Each printer space is 0.10 inch wide and each line occupies 1/6 inch of vertical space.)

5. Write a program to read in an array of N rows and M columns from cards with a reasonable number of data items per card (leaving space for sequence numbers). N and M, as well as a numeric representation of a date, are to be read from the first data card. Print the array in some sensible format, giving N and M and the data as part of the annotations in a suitable heading. Neither N nor M is to exceed 30. If this is violated in the data, print an error message.

CHAPTER 8 - INTRODUCTION TO PROCEDURES

8.1 SUBPROGRAM CONCEPT

There are many facets to the ALGOL concept of the subprogram. Only a few of them will be covered in this introductory chapter. Some of the more abstract (and powerful) aspects will be reserved for discussion in the next chapter.

A subprogram is a portion of a computer program which, for various reasons, is set apart from the main body of coding. It may be called into action from one or more points in the main program in such a way that, when the subprogram has been executed, control returns to the calling point. It is as if a transfer of control from the main program could carry with it a label from the next statement in sequence which could later be used to transfer back to that statement.

Such a capability exists in ALGOL in the form of procedure declarations and procedure statements. These will be formally defined later. For the present, we shall consider the procedure declaration to be a subprogram and a procedure statement a call on a subprogram.

The procedure statement causes a subprogram to be executed as if it existed as a body of ALGOL coding at the point occupied by the procedure statement.

A variety of reasons to use subprograms can be given. Examples:

1. If the same sequence of statements should occur at several points in a program, it can be made into a subprogram with gains in original writing and keypunching effort, ease of making revisions, readability, etc.

2. Subprograms can be devised and perfected in relatively simple test programs and then used in one or more, perhaps complex, unrelated programs. Little, if any, additional testing of the subprogram is needed with each new use.

3. Voluminous and difficult programming efforts can be packaged, as subprograms, for the use of many people other than their originators. With adequate description of the gross properties of a subprogram, it can be used by people who never need bother with understanding its details.

Note that in examples 2. and 3. there is no need to call upon a subprogram more than once in a given program. The payoff is obtained by being able to utilize with confidence a proven section of program without the slightest alteration. This is apart from the benefits derived from clarifying the logical structure of the main program and from the ability to make major revisions easily.

The present chapter provides only an introduction to the ALGOL procedure. Two important points will be established here:

1. The entire subprogram, called a procedure in ALGOL, has the status of a declaration in the program.

2. The form of procedure discussed here is called into action by an ALGOL statement which consists simply of the name arbitrarily given to the procedure in its declaration.

8.2 PROCEDURE DECLARATIONS AND STATEMENTS

The ALGOL constructs to be formally defined here are highly restricted relative to what is possible in B 5500 Extended ALGOL. The next chapter presents procedure declarations and statements in a much less restricted form.

As with any other identifier, it is essential to declare the meaning of a procedure identifier before it can be used in a procedure statement. Thus we have the procedure declaration which is defined as follows:

⟨procedure declaration⟩ ::= PROCEDURE ⟨procedure identifier⟩; ⟨procedure body⟩

⟨procedure identifier⟩::= ⟨identifier⟩

⟨procedure body⟩::= ⟨statement⟩

In this simplified version, we see that a PROCEDURE is declared in such a way as to associate an identifier with a body of ALGOL coding. The procedure identifier is formed by the same rules that apply to any identifier. The procedure body may consist of any of the many forms of unlabelled statements. In many cases it will be a block.

All identifiers appearing within a procedure body must, of course, be declared somewhere. If the procedure body is not itself a block, the identifiers occurring within it must be declared prior to the procedure declaration. If the procedure body is a block, additional "local" identifiers will be declared within it. Even then, under the restrictions of this chapter, some of the identifiers must be global (and declared prior) to the procedure declaration, in order for the procedure to have any meaning in the program in which it appears.

There is no restriction on the usage of identical identifiers inside and outside of a procedure for totally different purposes. Consider a variable, HAY, occurring both in a procedure and in the program in which the procedure is declared. If it is separately declared within the procedure body, the effect is the same as if it were spelled HAY in the program and HEY (or any other spelling) in the procedure body.

There is one additional restriction which applies to the procedure body of a procedure in B 5500 Extended ALGOL. Any identifier that is used but is not declared within a procedure body must be declared in a surrounding block, which is not itself another procedure body. This simply means that, if a procedure is declared within the body of another procedure, the body of the inner procedure may not use identifiers which are declared in the body of the outer procedure.

For the restricted form of the procedure being discussed here, the statement which calls a procedure into action is very simple. It consists solely of the procedure identifier. Formally, this is expressed by

⟨procedure statement⟩ : : = ⟨procedure identifier⟩

The net effect of the execution of a procedure statement in a program is the same as would be obtained if the entire procedure body were copied into the program at the point occupied by the procedure statement. Note, however, that such copying is not done by the ALGOL compiler. Rather the coding for the procedure body exists in one place only and all procedure statements cause control to be transferred to that one body of code. Upon completion of the execution of the procedure body, unless a go to statement has led to another point, control is returned to the point immediately following that procedure statement which called the procedure.

One small example will show how a simple procedure might be written and used. Suppose there were many occurrences in a program of a division with a (possible) divisor of zero. Suppose further that the desired results, C, for the division of A by B were specified by the table

A	B	C
(any)	$\neq 0$	A/B
>0	$=0$	+5@10
$\simeq 0$	$=0$	0
<0	$=0$	-5@10

These rules could be expressed in a single conditional statement and made into a procedure as follows:

```
PROCEDURE GOOFYDIV;

    IF B ≠ 0 THEN C ← A/B ELSE

    IF A ≃ 0 THEN C ← 0 ELSE

    IF A < 0 THEN C ← -5@10 ELSE

    C ← 5@10
```

(Note that we have ignored the availability of the common function SIGN.)

A typical use of this procedure would be to avoid anticipated divisions by zero in a statement, such as:

```
Y[I,J,K] ← W[I+1,J,K-F] / X[N-1,I]
```

In place of this, one would write

```
A ← W[I+1,J,K-F];

B ← X[N-1,I];

GOOFYDIV;

Y[I,J,K] ← C
```

Even in this simple-minded example, some saving in writing effort and a gain in clarity are found. Note, however, that the three variables A, B and C are global to the procedure body and must be declared before the procedure declaration.

8.3 ELEMENTARY USE OF THE PROCEDURE

Presented below is a complete but simple program that illustrates many features of the procedure which were formally discussed in the preceding section. This example should not be misconstrued as an attempt to "sell" a particular philosophy of input data formatting. It was chosen because it provides an interesting case for the use of procedures in ALGOL. A very similar example is used again in Chapter 9 to show, by comparison with the present example, the power of the more abstract features of procedures.

```
BEGIN

COMMENT     A PROGRAM TO ILLUSTRATE BASIC
            ASPECTS OF ALGOL SUBPROGRAMMING;

FILE IN     CARD (2, 10);

FILE OUT    LINE 1 (2, 15);

REAL        LOW, HIGH, DATAVAL, A,B,C,D,X;

INTEGER     LASTCDNO, RUN;

LABEL       EOFLAB, ERRLAB, START;

PROCEDURE   GETDATA;

            BEGIN

COMMENT     A DATA INPUT SUBPROGRAM THAT DOES
            THE FOLLOWING:

            (1) READS A DATA VALUE FROM A FILE
                "CARD",

            (2) PRINTS THE VALUE AND CARD NUMBER
                ON A FILE "LINE",

            (3) CHECKS FOR CORRECT CARD
                SEQUENCE,

            (4) CHECKS DATA VALUE FOR PROPER
                RANGE,

            (5) PRINTS ERROR MESSAGES AS NEEDED.

            BESIDES FILES "LINE" AND "CARD",
            SIX NON-LOCAL IDENTIFIERS ARE USED
            AS FOLLOWS:
```

```
                    LASTCDNO-PREVIOUS CARD NO.,
                    INITIALLY ZERO

                    DATAVAL - VALUE TO BE READ FROM
                              CARD

                    HIGH    - UPPER LIMIT ON DATAVAL

                    LOW     - LOWER LIMIT ON DATAVAL

                    ERRLAB  - LABEL TO GO TO UPON
                              ERROR

                     EOFLAB - LABEL TO GO TO UPON
                              END-OF-FILE
                  ALL VARIABLES MAY BE OF TYPE REAL;

FORMAT            CRD (I5, F20.9);

FORMAT            CIM("CARD NO.", I5, X10, "VALUE=",
                  F20.9),

                  SQER (/X50, "CARD SEQUENCE ERROR"),

                  VOR (/X50, "LAST VALUE OUT OF RANGE");

INTEGER           CARDNO;

LIST              KRD (CARDNO, DATAVAL);


                  READ (CARD, CRD, KRD) [EOFLAB];

                  WRITE (LINE, CIM, KRD);

                  IF CARDNO - LASTCDNO≠ 1 THEN

                BEGIN

                      WRITE (LINE, SQER); GO TO ERRLAB

                 END;

                      IF DATAVAL > HIGH OR DATAVAL < LOW
                      THEN

                BEGIN

                      WRITE (LINE, VOR); GO TO ERRLAB

                 END;

                      LASTCDNO ← CARDNO
```

```
                END OF GETDATA;

FORMAT          HEAD (X30, "THIRD DEGREE POLYNOMIAL ",
                "EVALUATION---PROGRAM WXYZ" ///),

                POLLY ("RUN NO." , I7, X6, "POLY=", E20.8/);

                WRITE (LINE, HEAD);

START:          LASTCDNO ← 0;

                HIGH ← 10000; LOW ← 1; GETDATA; RUN ←
                DATAVAL;

                HIGH ← 1  ; LOW ← 0; GETDATA; A ← DATAVAL;

                GETDATA; B ← DATAVAL; GETDATA; C ←
                DATAVAL;

                HIGH ← @ -6 ; LOW ← -.1; GETDATA; D ←
                DATAVAL;

                HIGH ← 100  ; LOW ← 10; GETDATA; X ←
                DATAVAL

                WRITE (LINE, POLLY, RUN,A + X × (B + X ×
                (C + X × D)));

                GO TO START;

ERRLAB:  EOFLAB:  END OF PROGRAM WXYZ.
```

The foregoing example program illustrates several important aspects of ALGOL procedures. It also contains several points of interest concerning ALGOL and good programming practice. The latter are flagged with asterisks in the list below.

1. The procedure named GETDATA is just one of many declarations in the program.

2. All identifiers which are global to the body of the procedure are declared prior to the occurrence of the procedure declaration,

3. The procedure body which can be any statement is, in this case, a block, since within the body there are declarations for CRD, CIM, SQER, VOR, CARDNO, KRD.

4. Exit from the body of the procedure is possible by four different paths, one via the action label, two via go to statements, and one via reaching the end of the body. The last one is the normal and most efficient method of exit.

*5. The procedure is reasonably well documented by means of a built-in comment. It could thus be used, with ease, in a variety of unrelated programs.

6. There are six separate calls on the subprogram GETDATA in the program. The effect is the same as if the body of the procedure were inserted into the ALGOL program at all six occurrences of the statement GETDATA.

*7. The good practice of providing a record in the program output of the input data has been followed here. Also, an identifying heading for the program output is furnished.

*8. The input data are qualitatively checked for correctness by the program. This process is somewhat simplified by the use of the subprogram GETDATA.

9. A point of interest is the use of two labels on the dummy statement preceding the final END of the program. In other programs using GETDATA these two labels might well be on different statements.

*10. The insertion of suitable comments after the END which terminates a procedure body contributes significantly to the readability of an ALGOL program

11. A final point of interest is that the only "computing" done in the entire program appears in the polynomial expression in the list of the write statement.

```
          BEGIN                                                                  DHT2  1
COMMENT         ILLUSTRATION OF THE SOLUTION OF THREE SIMULTANEOUS               DHT2  2
                EQUATIONS, USING TWO SUBPROGRAMS;                                DHT2  3
FILE IN         CARD (2,10);                                                     DHT2  4
FILE OUT        LINE 1(2,15);                                                    DHT2  5
FORMAT          ID ("ILLUSTRATION PROGRAM - CRAMERS RULE - DHT2"/),              DHT2  6
                HD (X25,"A",X15,"B",X15,"C",X15,"D",X35,"X"/),                   DHT2  7
                ANS (3(X15, 4F16.7, E26.6/)// ), ERR("DETERMINANT=0"/);          DHT2  8
FORMAT          KARD (4F15.7);                                                   DHT2  9
ARRAY           M[0:3,0:4], X[0:3];                                              DHT2 10
INTEGER ARRAY   Z, R1, R2, R3[0:6];                                              DHT2 11
REAL            D, DEN;                                                          DHT2 12
INTEGER         I, J, C1, C2, C3;                                                DHT2 13
LABEL           DUNN, REED, BAD, AUS;                                            DHT2 14
PROCEDURE       DETER3;                                                          DHT2 15
          BEGIN                                                                  DHT2 16
COMMENT         EVALUATES DETERMINANT WHOSE COLUMN NUMBERS IN                    DHT2 17
                THE M ARRAY ARE GIVEN BY C1, C2, AND C3;                         DHT2 18
INTEGER         I;                                                               DHT2 19
                D ← 0;                                                           DHT2 20
                FOR I ← 1 STEP 1 UNTIL 6 DO                                      DHT2 21
                D ← D + Z[I] × M[R1[I],C1] × M[R2[I],C2] × M[R3[I],C3]           DHT2 22
          END   OF DETER3;                                                       DHT2 23
PROCEDURE       KSET;                                                            DHT2 24
          BEGIN                                                                  DHT2 25
COMMENT         SETS UP Z,R1,R2,R3 ARRAYS WITH VALUES NEEDED BY DETER3;          DHT2 26
INTEGER         I;                                                               DHT2 27
                FOR I ← 1 STEP 1 UNTIL 6 DO Z[I] ← SIGN(I - 3.5);                DHT2 28
                R1[1] ← R2[2] ← R3[3] ← R1[4] ← R3[5] ← R2[6] ← 1;               DHT2 29
                R3[1] ← R1[2] ← R2[3] ← R2[4] ← R1[5] ← R3[6] ← 2;               DHT2 30
                R2[1] ← R3[2] ← R1[3] ← R3[4] ← R2[5] ← R1[6] ← 3                DHT2 31
          END   OF KSET;                                                         DHT2 32
                                                                                 DHT2 33
                WRITE (LINE, ID);           KSET;                                DHT2 34
REED:           READ (CARD, KARD, FOR I ← 1,2,3 DO                               DHT2 35
                                  FOR J ← 1,2,3,4 DO M[I,J])[DUNN];              DHT2 36
                C1 ← 1;  C2 ← 2;   C3 ← 3; DETER3;   DEN ← D;                    DHT2 37
                IF DEN = 0 THEN GO TO BAD;                                       DHT2 38
                C1 ← 4;                     DETER3;   X[1] ← D / DEN;            DHT2 39
                C1 ← 1;    C2 ← 4;          DETER3;   X[2] ← D / DEN;            DHT2 40
                       C2 ← 2;  C3 ← 4;     DETER3;   X[3] ← D / DEN;            DHT2 41
AUS:            WRITE (LINE, ANS, FOR I ← 1,2,3 DO                               DHT2 42
                                  [ FOR J ← 1,2,3,4 DO M[I,J], X[I]]);           DHT2 43
                GO TO REED;                                                      DHT2 44
BAD:            WRITE (LINE, ERR);  X[1] ← X[2] ← X[3] ← 0;  GO TO AUS;          DHT2 45
DUNN:     END   .                                                                DHT2 46
```

EXERCISES

1. Write a program whose main block contains only three statements which are procedure statements. These will be calls on 3 procedures to perform the following functions:

   Procedure 1: Read and print input data values for variables a, b, and c.

   Procedure 2: Compute the roots of the quadratic equation

   $$Y = ax^2 + bx + c$$

   Procedure 3: Print values for the roots.

2. Write a procedure called ERRORMESS which will print one of four error messages depending on whether the value of a global variable ERR is 1, 2, 3, or 4.

CHAPTER 9 - MORE ABOUT PROCEDURES AND BLOCKS

9.1 THE CONCEPT OF FORMAL PARAMETERS

Procedures of the kind discussed in Chapter 8 are often awkward to use. This is because the only means provided there, for communication of information (data) to and from the procedure, was via global identifiers. Further, the fact that these identifiers must be declared outside the procedure declaration hampers the possible utility of a given procedure, without change, in a variety of unrelated programs.

These difficulties are overcome by allowing another means of communication between a procedure and the program that calls it. In ALGOL, this is done by means of a parameter list. The procedure is written to operate on "formal parameters," which correspond to "actual parameters" furnished when it is called. The choice of identifiers to represent "actual parameters" need not have the least correspondence with those used for "formal parameters." Only one obvious requirement has to be met: formal and actual parameters must represent the same kinds of ALGOL entities. The latter point will be discussed at length in Section 9.4.

A concrete example will be used to preview the role of parameter lists before formally defining and discussing them in Sections 9.3 and 9.4. The simple procedure, GOOFYDIV, of Section 8.2 can be better declared as follows:

```
PROCEDURE GOOFYDIV (A, B, C);

REAL      A, B, C;

          IF B≠ 0 THEN C ← A/B ELSE

          IF A = 0 THEN C ← 0  ELSE

          IF A < 0 THEN C ← -5@10 ELSE C ← 5@10
```

If this is the case, then our previous use of this procedure is reduced to the writing of one statement:

```
GOOFYDIV (W[I+1,J,K-F], X[N-1,J], Y[I,J,K])
```

In place of the formal parameters A, B, C, we have used actual parameters which are the array elements W[I+1,J,K-F], X[N-1,J], Y[I,J,K]

The same net results are obtained as were obtained by the group of four statements in Section 8.2. Note, however, that the identifiers A, B and C are no longer needed as variables in the calling program. Rather, they are now formal parameters which are identifiers that are defined only within the procedure declaration. Their type is given in the REAL specification in the procedure heading. The latter two metalinguistic variables will be defined in Section 9.3.

As will be seen later, actual and formal parameters are not limited to identifiers associated with single numerical values. They may be (or represent) nearly any kind of ALGOL entity. We note that the read statement and the write statement actually have the form of procedure statements that possess actual parameter lists. Thus, they are often called "intrinsic procedures."

9.2 THE CONCEPT OF TYPED PROCEDURES

A further elaboration on the subprogram concept is found in ALGOL in the form of "typed procedures." These are most useful when the computations within the procedure body always yield a single numerical result. The type (REAL or INTEGER) of this result dictates the type given to the procedure declaration. The presence of a type before the reserved word procedure indicates that this subprogram will be called as a special kind of function designator. Thus, an expanded definition of a function designator, over that given in Section 2.7, will be discussed in Section 9.5.

Once again, the example of Section 8.2 can be used to provide a more concrete preview of typed procedures. Since the subprogram GOOFYDIV yields only one (REAL) result, it can be declared as follows:

```
REAL PROCEDURE GOOFYDIV (A, B);
REAL      A, B;
          IF B ≠ 0 THEN GOOFYDIV ← A/B ELSE
          IF A = 0 THEN GOOFYDIV ← 0   ELSE
          IF A < 0 THEN GOOFYDIV ← -5@10 ELSE
          GOOFYDIV ← 5@10
```

With this kind of declaration, the procedure is "called" much as we might "call" the SQRT common function:

```
Y[I,J,K] ← GOOFYDIV (W[I+1,J,K-F],X[N-1,J])
```

And once more, the same results are obtained.

It is not essential that communication with a typed procedure be via a parameter list. Some or all information can be passed by the mechanism of global identifiers. (This is also true of non-typed procedures.) In fact, although the typed procedure is designed to transmit a single numerical result in a convenient way, there is nothing wrong with allowing it to produce many "results" in the form of changes in globally declared variables, or as output from the program.

9.3 PROCEDURE DECLARATIONS - REDEFINED

Sufficient introduction and previews have now been given to allow an intelligible formal definition of procedure declarations. The definition given in Section 8.2 will be seen to be a compatible subset of that to be presented here. This version is wholly consistent with the subset of B 5500 Extended ALGOL being presented throughout this text.

A procedure is formed in either of two ways, as follows:

```
⟨procedure declaration⟩ : : = PROCEDURE ⟨procedure heading⟩
                               ⟨procedure body⟩ | ⟨type⟩
                               PROCEDURE ⟨procedure heading⟩
                               ⟨procedure body⟩
```

The chief difference between this definition and that given in Section 8.2 lies in the procedure heading which is defined and discussed in detail below. One other apparent difference is in the possibility of a type appearing before the reserved word procedure. The definition of procedure body is unchanged from that given in Section 8.2. For the convenience of the reader we repeat the definitions.

⟨procedure body⟩ ::= ⟨statement⟩

⟨type⟩ ::= REAL | INTEGER

From the discussion in Section 9.2, it should be apparent that the existence of a type on a procedure declaration puts it into a special category. The manner of calling it is discussed in Section 9.5. Further, the procedure body must contain and cause to be executed at least one assignment statement, with the name of the procedure to the left of the assignment operator. Although it may appear that this would also allow the procedure name to be used elsewhere in the procedure body as a simple variable, such is not the case.

One other rule must be observed within the procedure body of a typed procedure: exit must not be made via a go to statement.

To complete the definition of a procedure declaration, we now have only to define the procedure heading.

⟨procedure heading⟩ ::= ⟨procedure identifier⟩
⟨formal parameter part⟩;
⟨specification part⟩

⟨procedure identifier⟩ ::= ⟨identifier⟩

⟨formal parameter part⟩ ::= ⟨empty⟩ | (⟨identifier list⟩)

⟨identifier list⟩ ::= ⟨identifier⟩ | ⟨identifier list⟩, ⟨identifier⟩

⟨specification part⟩ ::= ⟨empty⟩ | ⟨specification list⟩

Definition of specification list will be pursued below. Note that, since both the formal parameter part and the specification part can be empty, a valid procedure heading could be simply an identifier followed by a semicolon. This agrees with the definitions of Section 8.2.

Note also that the semicolon is a necessary part of the procedure heading.

If the formal parameter part is not empty, it can only be one or more identifiers separated by commas and enclosed in parentheses. These identifiers are simply called formal parameters. They may represent a variety of ALGOL entities, and the purpose of the specification part is to "specify" the kind of entity represented by each formal parameter. In no case is a formal parameter anything other than an identifier standing alone. (It makes no sense to place constants or subscripted variables in a formal parameter list, for example.) Because of the previously mentioned restriction on leaving a typed procedure body via a go to statement, a formal parameter of a typed procedure may not be a label.

The necessity of a specification list can be seen if one recalls that, in B 5500 Extended ALGOL, it is essential that all identifiers be "known" to the compiler prior to any use of them in an ALGOL program. The mere appearance of an identifier as a formal parameter is not sufficient to allow its use. The compiler must know a little more about it. The manner in which this is done is shown in the definitions to follow:

⟨specification list⟩ ::= ⟨specification⟩; | ⟨specification list⟩ ⟨specification⟩;

⟨specification⟩ ::= ⟨specifier⟩ ⟨identifier list⟩ | ⟨array specification⟩

⟨specifier⟩ ::= ⟨type⟩ | FILE | LIST | FORMAT | LABEL

⟨array specification⟩ ::= ARRAY ⟨array specifier list⟩ | ⟨type⟩ ARRAY ⟨array specifier list⟩

⟨array specifier list⟩ ::= ⟨array specifier⟩ | ⟨array specifier list⟩, ⟨array specifier⟩

⟨array specifier⟩ ::= ⟨array identifier list⟩ [⟨lower bound list⟩]

⟨array identifier list⟩ ::= ⟨identifier list⟩

⟨lower bound list⟩ ::= ⟨specified lower bound⟩ | ⟨lower bound list⟩, ⟨specified lower bound⟩

⟨specified lower bound⟩ ::= ⟨integer⟩ | *

The student might easily confuse the above specifications with declarations, since they do serve similar purposes. However, only in the case of simple variables and labels do the specifications and declarations appear identical. Except for array specifications, nothing more than the specifier is used to give information about a formal parameter. Declarations which are local to the procedure body must not be confused with the above specifications. Specifications appear in the procedure heading and declarations in the block head of the procedure body.

The array specification provides information as to the kind of array represented by the formal parameter. It also contains information concerning the lower bounds of the actual array that will be "passed" to the procedure when it is called. If a lower bound is constant, it is specified as an integer. If it is unknown, or may vary from call to call, an asterisk is used to impart this information. Since a lower bound of zero is most efficient in terms of running speed on the B 5500, one finds a high percentage of library procedures written with specified lower bounds of zero for formal parameters that are array identifiers.

Only one simple example will be given here since several others are available for examination in Sections 9.1, 9.2, and 9.6. The present example is a typed procedure for computing the factorial of an Integer N.

```
INTEGER PROCEDURE FACT (N); INTEGER N;
        BEGIN
INTEGER         PROD, I;
                IF N < 0 THEN FACT ← 0 ELSE
        BEGIN
                PROD ← 1;
                FOR I ← 1 STEP 1 UNTIL N DO
                PROD ← PROD x I; FACT ← PROD
        END
        END OF FACT
```

A point of interest here is the manner in which the correct value for FACT(0) is obtained.

9.4 PROCEDURE STATEMENTS - REDEFINED

A procedure statement is formed by the following rules:

⟨procedure statement⟩ ::= ⟨procedure identifier⟩ ⟨actual parameter part⟩

⟨actual parameter part⟩ ::= ⟨empty⟩ | (⟨actual parameter list⟩)

⟨actual parameter list⟩ ::= ⟨actual parameter⟩ | ⟨actual parameter list⟩ , ⟨actual parameter⟩

⟨actual parameter⟩ ::= ⟨arithmetic expression⟩ | ⟨array identifier⟩ | ⟨file identifier⟩ | ⟨format identifier⟩ | ⟨list identifier⟩ | ⟨label⟩

Note that if the actual parameter part is empty, we have exactly the form given for the procedure statement given in Section 8.3.

A procedure statement causes a procedure body which has been previously given in a procedure declaration, to be called for execution. If that procedure declaration had a formal parameter part,

the procedure statement must have an actual parameter part which provides the actual parameters to be used during this call on the procedure. A one-for-one correspondence must exist between the actual parameters in the actual parameter part and the formal parameters which appear in the formal parameter part of the procedure declaration; this correspondence is one of position, where the position of an actual parameter given in the procedure statement corresponds to the position of a formal parameter in the procedure declaration.

A general description of the operation of the procedure statement can be given as follows:

1. The formal parameters are replaced, wherever they appear in the procedure body, by the corresponding actual parameters. Identifiers thus introduced into the procedure body may be identical to local identifiers already there. Each is handled in such a way, however, that no conflict occurs.
2. The procedure body, when modified as stated above, is then entered.
3. Upon completion of the computations within the procedure body, the program flow returns to the statement following the (calling) procedure statement.

The actual parameters may be expressions and array, file, format, label and list identifiers. The actual expression or pertinent identifier of the actual parameter replaces the corresponding formal parameter wherever it appears in the procedure body. If a formal parameter appears to the left of an assignment operator in the procedure body, the corresponding actual parameter may only be a variable.

If a simple variable is an actual parameter, the corresponding formal parameter is replaced wherever it appears in the procedure body, by the identifier of the simple variable. The value represented by the simple variable is referred to each time the variable is encountered during the execution of the procedure body. The actual parameter is accessible throughout the procedure and can therefore have its value altered.

Where a subscripted variable is an actual parameter, it is placed in the procedure body wherever the corresponding formal parameter appears. The subscript expression remains intact, and is evaluated each time the subscripted variable is referred to during the execution of the procedure body.

Where the actual parameter is a function designator, the corresponding formal parameter is replaced by the function designator wherever the formal parameter appears in the procedure body. The function designator is evaluated whenever it is encountered during the course of execution of the procedure body.

In the case where an arithmetic expression is an actual parameter, the corresponding formal parameter is replaced by the expression in question. This expression is evaluated whenever it is encountered during the execution of the procedure body.

When the actual parameter is an array identifier, the corresponding formal parameter is replaced by the array identifier wherever the formal parameter appears in the procedure body. Any appearance of the formal parameter in the procedure body makes reference to the actual array designated by the array identifier.

Where a file, format, or list identifier has been given as an actual parameter, the corresponding formal parameter is replaced by the identifier of the actual parameter wherever the formal parameter appears in the procedure body. I-O statements in a procedure body can thus be caused to utilize arbitrary files, formats and lists which have been declared outside the procedure body being executed.

A simple example of a procedure statement was given in Section 9.1; a more elaborate example is presented in Section 9.6.

There is a useful feature of ALGOL procedure declarations that is not included in the subset being presented here. That is, the concept of call-by-value, which is used heavily in most B 5500 ALGOL library procedures. Call-by-value is discussed with clarity in most of the ALGOL reference materials.

9.5 FUNCTION DESIGNATORS - REDEFINED

We will here expand the definition of a function designator given in Section 2.7 as follows:

⟨function designator⟩ ::= ⟨procedure identifier⟩
⟨actual parameter part⟩|
⟨common function identifier⟩
((⟨arithmetic expression⟩))

⟨procedure identifier⟩ ::= ⟨identifier⟩

⟨actual parameter part⟩ ::= ⟨empty⟩|(⟨actual parameter list⟩)

⟨actual parameter list⟩ ::= ⟨actual parameter⟩|
⟨actual parameter list⟩,
⟨actual parameter⟩

⟨actual parameter⟩ ::= ⟨arithmetic expression⟩|⟨array identifier⟩|
⟨file identifier⟩|⟨format identifier⟩|
⟨list identifier⟩

Note that the actual parameter part as defined above is the same as that given for procedure statements in Section 9.4 except that it may not contain a label.

The only kind of procedure that may be called as a function designator is one which was declared with a ⟨type⟩ in front of the reserved word PROCEDURE. Upon completion of the execution of the procedure body, control returns to the evaluation of the arithmetic expression in which the function designator was encountered. With the exception of the latter action, all other aspects of the operation of a function designator are the same as those given for procedure statements in Section 9.4.

Examples of the use of a function designator calling on a typed procedure are given in Sections 9.2 and 9.6. The rules for using any kind of function designator are given in Section 2.7.

9.6 EXAMPLE OF USE OF PARAMETER LISTS

The example program below is supposed to accomplish the same thing as the program in Section 8.3. It is designed to illustrate the manner in which parameter lists are used in practice.

```
        BEGIN

COMMENT         A PROGRAM TO DEMONSTRATE THE USE
                OF PROCEDURE PARAMETER LISTS;

INTEGER         LASTCDNO, RUN;

REAL PROCEDURE  POL3(U,A0,A1,A2,A3);

REAL            U,A0,A1,A2,A3;
                POL3 ← A0+U × (A1+U × (A2+U × A3));

PROCEDURE       GETDATA (FIN,FOUT, LEOF, LERR,
                H, L, D);

REAL            H, L, D;

FILE            FIN, FOUT;

LABEL           LEOF, LERR;
```

```
BEGIN

COMMENT          A PROCEDURE FOR DATA INPUT WHEREIN:

                 I    DATA ARE READ FROM A FILE "FIN",
                 II   THE DATA VALUES ARE WRITTEN ON
                        A PRINTER FILE "FOUT",
                 III  CARD SEQUENCE IS CHECKED AGAINST
                        A GLOBAL VARIABLE "LASTCDNO",
                        INITIALLY ZERO,
                 IV   EXIT IS TO LEOF AT END-OF-FILE,
                 V    EXIT IS TO LERR IF EITHER D < L OR
                        D > H, LIMITS GIVEN BY CALLING
                        PROGRAM;

FORMAT           CRD (I5, F20.9),
                 CIM ("CARDNO.", I5, X10, "VALUE = ",
                    F20.9),
                 SQER (/ X50, "CARD SEQUENCE ERROR"),
                 VOR (/X50,
                 "LAST VALUE OUT OF RANGE");

INTEGER          CARDNO;

LIST             KRD (CARDNO,D);
                 READ (FIN, CRD, KRD)[LEOF];
                 WRITE (FOUT, CIM, KRD);
                 IF CARDNO - LASTCDNO ≠ 1 THEN

                 BEGIN

                        WRITE (FOUT, SQER); GO TO LERR

                 END;

                        IF D > H OR D < L THEN

                 BEGIN

                        WRITE (FOUT, VOR); GO TO LERR

                 END;

                        LASTCDNO ← CARDNO

                 END OF GETDATA;

REAL             A, B, C, D, X;

LABEL            EOFLAB, START;

FILE IN          CARD (2, 10);
```

```
FILE OUT        LINE 1 (2,15);

FORMAT          HD (X30, "THIRD DEGREE POLYNOMIAL",
                  "EVALUATION- - - PROGRAM ZYXW"
                  ///),
                POLY ("RUN NO.", I17, X6, "POLY =",
                  E20.8);
                WRITE (LINE, HD);

START:          LASTCDNO ← 0;
                GETDATA (CARD, LINE, EOFLAB,
                  EOFLAB, @5, 1, RUN);
                GETDATA (CARD, LINE, EOFLAB,
                  EOFLAB, 1, 0, A);
                GETDATA (CARD, LINE, EOFLAB,
                  EOFLAB, 1, 0, B);
                GETDATA (CARD, LINE, EOFLAB,
                  EOFLAB, 1, 0, C);
                GETDATA (CARD, LINE, EOFLAB,
                  EOFLAB, @-6, -.1, D);
                GETDATA (CARD, LINE, EOFLAB,
                  EOFLAB, 100, 10, X);
                WRITE (LINE, POLY, RUN, POL3
                  (X,A,B,C,D));
                GO TO START;

EOFLAB:  END OF PROGRAM ZYXW.
```

No claim is made that the above example is a sensible program. It does, however, demonstrate the use of parameter lists in procedure declarations, procedure statements and function designators. Points of interest:

1. Both POL3 and GETDATA are declared in the block head of the program, along with the rest of the declarations.

2. POL3 is lacking any comments for documentation. This is perhaps excusable since the procedure body of POL3 is just one assignment statement, while the body of GETDATA is a block.

3. No conflict arises over the existence of the identifier D as a simple variable in the program and also as a formal parameter of GETDATA.

4. GETDATA uses both formal parameters and a global variable to pass information in and out.

5. Three different kinds of formal parameters are used in the procedure GETDATA.

6. The specifier for the file identifiers is merely FILE, without IN or OUT attached.

7. The list of the write statement contains two arithmetic expressions, RUN and POL3 (X, A, B, C, D). The latter is a function designator calling on the typed procedure POL3.

8. If actual use of this procedure were ever contemplated, the programmer would probably prefer to avoid making the file identifiers formal parameters of GETDATA. In that case, the declarations for CARD and LINE would have to occur before the declaration of GETDATA. Also, the procedure body would have to be changed to use CARD in place of FIN and LINE in place of FOUT.

9.7 BLOCKS IN ALGOL

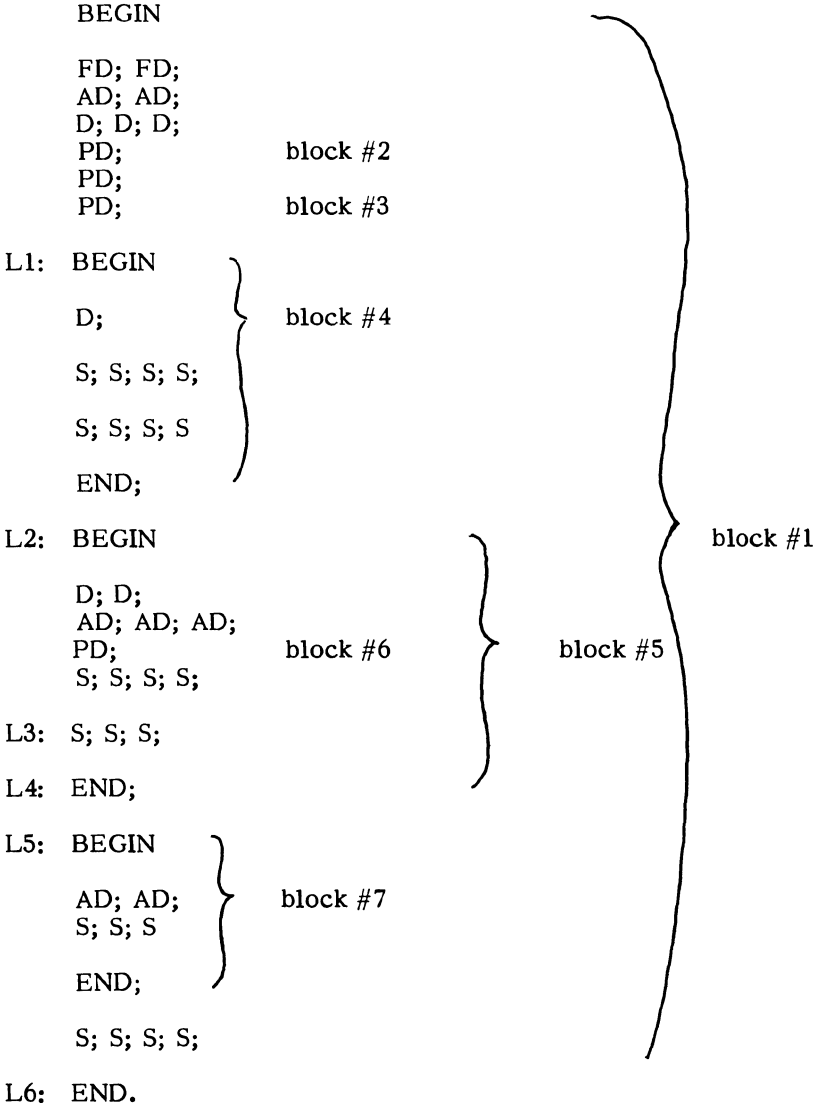
The block is a powerful feature of ALGOL. It is particularly important in large programs and in many programs that contain procedure declarations. This section will supplement the discussion of blocks in Chapter 3, in the light of concepts covered in the later chapters. If Chapter 3 has not been reviewed since it was first encountered, the student is advised to do so at this point.

Blocks have two primary purposes. One is that of isolating portions of a program from one another so that identifiers can be freely chosen in each of them. This is especially important when a block is a procedure body because it makes possible the development of libraries of procedures which can be used in any program. No conflict results when the same identifier is declared to have dissimilar meanings in different blocks of a program. This is true whether the blocks are procedure bodies or not. Major portions of a large program can be written separately, by different people, with coordination on the choice of only the global identifiers.

The other prime use of the block is to obtain automatic segmentation of a program in order to cope with the realities of finite computer memories. With a little care, a program can usually be written such that portions which may be inactive for appreciable periods of time become separate blocks. In the B 5500 these are compiled as program segments which may be kept in auxiliary storage (such as a drum or disk) whenever they are not active. In this way a maximum amount of core memory is made available for data. This also provides a mechanism for minimizing the amount of memory used for data storage at any given time. Memory is allocated to the storage of an array, only within the block in which that array is declared. When a block is exited, all memory used for arrays declared in that block is released for other uses.

The example to follow will illustrate many of these points. For clarity and brevity, several abbreviations will be used:

AD - array declaration
FD - file declaration
PD - procedure declaration
D - any other declaration
S - statement



This is a schematic representation of the general form of many scientific programs. Block #1 is, of course, the program itself. Blocks 2, 3 and 6 are the bodies of procedure declarations. Block #4 might contain statements that read the data, perform some preliminary data conversions, print a summary of the input data and initialize some arrays that are declared in the block head of block #1. Block #5 might perform a lengthy iterative computation, perhaps using a procedure (block #6) to do matrix inversions. Final processing of the results and the printing of same might be conducted in block #7.

We note that, while one of blocks 4, 5 or 7 is active, the other two need not be present in memory. Blocks 2 and 3 might be activated by procedure calls anywhere, but unless they are being called frequently there is no need to have them in memory. The procedure declared in block #5 may be called only from within that block since it is local to the block. When block #5 is entered, memory is allocated for the three arrays declared therein. When block #5 is exited, this memory is released and becomes available for allocation to the arrays declared when block #7 is entered. It is again released after execution of block #7. One of the four statements at the end may cause control to return to block #4 or #5. Since a block must always be entered through the block head, a return to block #4 or #5 can only be effected with a go to statement referencing the label L1 or L2, respectively.

The above example also provides an illustration of the local nature of label declarations. The labels L1, L2, L5 and L6 must be declared in the block head of block #1. Labels L3 and L4 must be declared in block #5. The statement GO TO L3 cannot be used anywhere but in block #5. However, a transfer to L1, L2, L5 or L6 can be made from anywhere in the program since they are local only to the block which constitutes the program.

Finally, a word of caution must be given on running speed considerations in the use of blocks. Rapid looping through a block is not desirable if that block contains array declarations. The time required by the computer to allocate and release memory for array storage is not insignificant unless a fair amount of computing is performed within that block. The most efficient way to exit any block is by falling out of the end of it.

```
          BEGIN                                                          DHT3  1
COMMENT         ILLUSTRATIVE USE OF PROCEDURES WITH AND WITHOUT          DHT3  2
                PARAMETER LISTS IN COMPUTING AN APPROXIMATION OF SIN(X)}  DHT3  3
FILE OUT        LINE 1 (2,15)}                                           DHT3  4
FORMAT          HEADING (X40,"SIN(X) COMPARISON PROGRAM - DHT3" ///)}    DHT3  5
REAL            PHI, FIE, SY}                                            DHT3  6
REAL PROCEDURE  SYGNE (A)}                                               DHT3  7
REAL            A}                                                       DHT3  8
          BEGIN                                                          DHT3  9
COMMENT         A POLYNOMIAL APPROXIMATION OF SIN(A)}                    DHT3 10
REAL            COEF, S, T, TSQ}                                         DHT3 11
                S ← -.00436248 }  T ← A / 90}   TSQ ← T × T}             DHT3 12
                FOR COEF ←.07948766, -.64592098, 1.57079485 DO           DHT3 13
                              S ←COEF + TSQ × S}                         DHT3 14
                SYGNE ← S × T                                            DHT3 15
          END   OF SYGNE}                                                DHT3 16
REAL PROCEDURE  GOODANGLE}                                               DHT3 17
                FOR FIE ← ABS(PHI + 90) STEP -360 UNTIL -180 DO          DHT3 18
                GOODANGLE ← ABS(FIE) - 90}                               DHT3 19
PROCEDURE       GETBUSY}                                                 DHT3 20
          BEGIN                                                          DHT3 21
FORMAT          COLLABS (" X (DEGREES)   SYGNE(X)          SIN(X)" /),   DHT3 22
                WAY1 (I8, 2F16.7)}                                       DHT3 23
LIST            RSLTS (PHI, SY, SIN (PHI / 57.29578))}                   DHT3 24
                WRITE (LINE, HEADING)}    WRITE (LINE, COLLABS)}         DHT3 25
                FOR PHI ← 90 STEP -15 UNTIL -150, 0 STEP 36 UNTIL 756 DO DHT3 26
          BEGIN                                                          DHT3 27
COMMENT         SYGNE (GOODANGLE) IS A FUNCTION DESIGNATOR WHOSE ACTUAL  DHT3 28
                PARAMETER IS AN EXPRESSION CONTAINING ANOTHER FUNCTION   DHT3 29
                DESIGNATOR, GOODANGLE, WHICH HAS NO PARAMETER LIST}      DHT3 30
                SY ← SYGNE (GOODANGLE)}                                  DHT3 31
                WRITE (LINE, WAY1, RSLTS)                                DHT3 32
          END   PHI LOOP                                                 DHT3 33
          END   OF GETBUSY}                                              DHT3 34
COMMENT         THE ONLY STATEMENT IN THE MAIN PROGRAM BLOCK FOLLOWS     DHT3 35
                GETBUSY}                                                 DHT3 36
          END   OF PROGRAM.                                              DHT3 37
```

EXERCISES

1. Re-work exercise 6, Chapter 4, using a REAL procedure to compute the square root.

2. Write a procedure to find the minimum and maximum values of the elements in a single dimensioned array. The formal parameter list should include the array identifier and two simple variables for the minimum and maximum values. The elements of interest have subscripts 1 thru N, where the value of N is provided in the zeroeth element.

3. Re-work the Chapter 8 example program on the solution of three simultaneous equations. Incorporate the solution process into a procedure named CRAMER which has arrays M and X for formal parameters. Nest KSET and DETER3 inside the body of CRAMER. Make the arrays R1, R2 and R3 global to CRAMER. Make DETER3 a REAL procedure with formal parameters C1, C2 and C3, such that the actual parameters for the four calls on DETER3 would be (1,2,3), (4,2,3), (1,4,3), and (1,2,4).

CHAPTER 10 - DIAGNOSTICS

10.1 ERRORS IN PROGRAMMING

There are two broad classes of errors that occur regularly during the development of a digital computer program. One class arises from violations of the syntax of the programming language. Errors of this type make it impossible for the compiler to complete the conversion of the source program into machine language. The other class covers logical faults in the program that prevent its successful execution. Logical errors may precipitate a variety of error conditions, some of which force an automatic termination of the program, while others merely cause incorrect or incomplete results.

Some logical errors are very easily traced. Others produce only symptoms, at remote points, that may require considerable effort to diagnose. Specific rules to follow in debugging any program are impossible to formulate. However, a number of general techniques are available. These will be discussed in this chapter.

Unfortunately, both syntactical and logical difficulties are often caused by a single type of error, that of incorrect transcription of a program into a deck of punched cards. Some of these are simple typographical errors which every human makes with some regularity. Many result from illegibly or incorrectly drawn characters. Most computer installations establish conventions that permit distinction of easily confused character pairs such as 1 and I, 2 and Z, 0 (zero) and O, X and X. Coding forms are usually designed to promote accurate card punching. Seasoned programmers take every precaution possible to insure the correctness of their card decks. Often this includes visual verification of a listing of their program.

A common problem with card decks is that of getting one or more cards out of sequence. Most installations utilize a scheme of sequence numbering that is punched somewhere in columns 73

through 80 of every card. These can be checked quite easily, either visually or via a service program. The compiler normally ignores these card columns completely. Thus, a card sequence error can go undetected until (or unless) it causes either a syntactical or logical difficulty.

Most computer users ultimately conclude that they themselves are responsible for the majority of the errors that occur in their programs. This does not prohibit their use of a machine, the computer itself, to systematically search out these errors. Indeed, in many programs, it becomes practically impossible to find all faults without use of the computer. Inadvertent programming errors and erroneous numerical methods can produce equally baffling symptoms. Often it is necessary to build self-checking features into a complex program to detect errors or to guard against a future case of bad data or data that bring to light a long undiscovered program error. Programmer ingenuity plays an important role here.

In the sections to follow, those features of B 5500 Extended ALGOL that aid in debugging will be discussed. These and the normal output statements are essentially the only debugging tools at the disposal of the programmer. The general technique of using such tools is commonly called "symbolic debugging." It is usually more efficient, for both man and computer, than the machine-oriented techniques of console debugging and memory dumping. In fact, to debug an ALGOL program on the B 5500 by the latter approach is a near impossibility. Moreover, the use of symbolic methods of debugging is totally consistent with the use of ALGOL as a programing language.

10.2 SYNTACTICAL ERROR DIAGNOSTICS

An ALGOL program must be free of syntactical errors before it can be successfully translated into machine instructions by the compiler. Since errors are not entirely avoidable, the compiler is designed to supply messages concerning any constructs that it is unable to translate.

Error messages are given in the form of an error code and a reference to that entity which appears to be incorrect. For example, an identifier which was not declared or which is misspelled is meaningless to the compiler. There is no way to correctly translate a statement in which such an identifier occurs. Note that a misspelled reserved word must be assumed to be an identifier (which may or may not have been declared). Error messages pointing out undeclared identifiers occur rather frequently.

In general, any violation of the syntax rules of ALGOL will cause diagnostic error messages to be given. The message will reference

that entity in the program which gave the first indication of a syntax violation. Often the actual error will be found at some distance to the left of that entity (considering the program as one long string that is read from left to right).

For example, the syntactical inconsistency of

```
T ← ABS ((I-J/(I+J)+8×T)+ A*2-COS(PHI);
```

is not detectable until the compiler encounters the semicolon. It then refers to the semicolon and provides the error code for the message

MISSING RIGHT PARENTHESIS.

It is unable to determine that the parenthesis was omitted between the J and the slash, since one could legally occur in several places.

In some cases the message can only tell what seems to be syntactically wrong, since the actual error may not be syntactically incorrect in itself. For example, suppose I- were punched for IM. Then,

```
FOR I ← 1 STEP 1 UNTIL I- DO etc.
```

results in referencing the DO and the message

PRIMARY MAY NOT BEGIN WITH A QUANTITY OF THIS TYPE.

Here is another place where a knowledge of ALGOL syntax is vital to the programmer. When the cause of an error message is not immediately obvious, the programmer may need to use the syntax to deduce the cause. Since he can regard the compiler as a perfect embodiment of the syntax rules, he can work backward from the diagnostics it provides.

It can happen that, after encountering one or more errors, the compiler will appear to become confused and begin emitting spurious error diagnostics. These need not be troublesome if the programmer proceeds as follows:

1. Find and correct the error that caused the first message. This message is never spurious.

2. Attempt to find the error that caused the next message. Correct any error that is found. If one is not readily found, pass on to step 3.

3. Repeat step 2 for all succeeding error messages.

4. Even if some of the indicated errors below the first one are not identified, compile the program again and repeat these steps until all syntax errors are corrected.

There is a good reason for following these steps, rather than insisting on finding and correcting all indicated errors. This has to do with the manner in which the compiler processes the program. The program string is scanned from left to right, and every basic element of the language is immediately translated if it fits correctly into the syntax. At the first violation of syntax, an error message describing the violation is issued. Depending on the type of violation, the compiler may be forced to skip ahead to the next semicolon, where a new statement or declaration begins, in order to restart its analysis of the program string. Such a skip may cause a syntactically vital program element (such as a BEGIN or an END, or the latter portion of a declaration) to be ignored. This precipitates the same series of error messages that would appear, if the skipped portion of the program were inadvertently omitted by the programmer. Thus, some of the error diagnostics after the first one may be spurious in that they can be a consequence of a previous error. In most cases, spurious messages do not persist beyond the first or second attempt to compile an ALGOL program.

A final comment is in order on the role of the semicolon in ALGOL. The presence of an extraneous semicolon can cause syntactical or logical errors that are sometimes obscure unless the rules of syntax are carefully applied. Recall that the semicolon is classed as a separator which is used to separate declarations from declarations, declarations from statements, and statements from statements. Recall further the existence of the dummy statement, which is nothing but empty space. Often an extraneous semicolon creates an unwanted dummy statement.

The following illustrations will complete this discussion.

```
FOR I ← 0 STEP 1 UNTIL 40 DO; A [I] ← 1
```

There is no syntactical error here, but, at run time, there would be 41 executions of the dummy statement between the DO and the semicolon. The assignment statement would be executed only once.

```
BEGIN;D;D;S;S END
```

Here there is a syntactical error, namely, that of a statement preceding a declaration in a block. The offending statement is the dummy statement between the BEGIN and the semicolon.

```
BEGIN D;;D;S;S END
```

This extra semicolon causes the same syntactical error as in the previous case.

```
IF BE THEN BEGIN S;S END; ELSE S
```

The semicolon after the END is syntactically wrong because it causes ELSE to appear as the beginning of an illegal statement.

10.3 RUN-TIME ERROR CONDITIONS

Any attempt on the part of a program to exceed computer hardware limitations is necessarily categorized as a run-time error. Such an attempt is usually only the first symptom of a logical error in the program because actual machine errors are very rare and machine limitations are quite easy to live within. Therefore, an analysis of the cause of any such event will usually result in the elimination of at least one error in the program. The analysis itself may require use of techniques to be discussed in Section 10.4

The number and nature of hardware restrictions vary from one computer to another. Four important arithmetic errors are detected by the B 5500 computer. A message, identifying the error and the point in the ALGOL program where it occurred, is provided to the programmer. Such information is usually essential to begin a search for the program error.

The four arithmetic errors are described as follows:

1. Divide by Zero. Any attempt to perform a division by zero.

2. Integer Overflow. Any attempt to assign a value to an INTEGER variable that is outside the allowed range, i.e., -549755813887 to +549755813887.

3. Exponent Overflow. Any attempt to compute a result that is greater than the maximum allowed for the absolute value of a REAL quantity. This maximum is (8*13-1) x 8*63, or approximately 4.3@68. (The lower limit is 8* (-51) or roughly 1.1@-47. An attempt to compute a smaller value, other than zero, produces an exponent underflow condition which is not usually considered an error. A value of zero is normally supplied as the result in such a case.)

4. Invalid Index. Any attempt to use an index for a subscripted variable that lies outside the declared range for that subscript. For example, assuming the declaration for A is

   ```
   ARRAY   A [-5:10,0:250 ]
   ```

   an invalid index would occur if an attempt were made to utilize the variable A [I,J] when I and J have values such that I>10 or I < -5 or J >250 or J< 0.

This feature not only helps to pinpoint program errors but also protects a program from the errors of another program when they are executed concurrently on the same computer.

10.4 USE OF MONITOR AND DUMP

Often the actual cause of an arithmetic error is not immediately obvious. It may become obvious only after considerable detective work. Usually this requires one or more additional attempts to execute the program with modifications designed to output intermediate results of some sort.

Such additional output can be programmed using the write statement. However, it is usually more desirable and much simpler to "program" this output with the diagnostic declarations, monitor and dump, which are described below. With these, it is very easy to study the behavior of a program in any suspected area. One or more critical variables can be monitored to show the manner in which they change. The logical flow can be depicted in terms of the sequence in which labels are encountered. Periodic "pictures" can be printed of as many of the program variables as may be desired. Used in combination, these techniques are powerful enough to furnish the information needed to isolate even the most obscure errors. They allow the programmer to use the machine itself to "debug" his program without requiring any more acquaintance with the machine than was needed to write the ALGOL program in the first place.

The exact type and quantity of diagnostic output needed to find a given error depend heavily on the kind of error and on the background of the programmer. Since the error is one of program logic (as opposed to a violation of ALGOL syntax), it is necessarily quite problem-dependent. Two requirements are thus placed on the programmer:

1. He must be prepared to improvise diagnostic approaches that vary with both the problem and the error symptoms.

2. To find and correct all of his own errors, he must be intimately familiar with his problem and the algorithm being used to solve it.

This presupposes sufficient knowledge of ALGOL to use it correctly as a programming language. The previous nine chapters were devoted to assistance in meeting this requirement.

We close this chapter with discussions of the syntax and the effects produced by the monitor and dump declarations. It will be noted, that like the list declaration, these declarations reference identifiers which must have been previously declared. Thus, it is common to place monitor and dump declarations at the very end of a block head (just before the first statement of that block). Any labels referenced by these declarations must be declared in that same block head, since labels are always local in scope. Variables are not subject to this restriction. These declarations fit into the syntax as follows:

⟨diagnostic declarations⟩ ::= MONITOR ⟨monitor part⟩ |
DUMP ⟨dump part⟩

The monitor declaration will be discussed first. The composition of the monitor part is specified by the following rules:

⟨monitor part⟩ ::= ⟨file identifier⟩ (⟨monitor list⟩)

⟨monitor list⟩ ::= ⟨monitor list element⟩ | ⟨monitor list⟩, ⟨monitor list element⟩

⟨monitor list element⟩ ::= ⟨simple variable⟩ | ⟨label⟩ | ⟨array identifier⟩ | ⟨subscripted variable⟩

In most cases, the file identifier will be that associated with the printer in a previous file declaration. In fact it is usually the same file used by the program to produce normal printer output.

The monitor declaration causes certain entities to be placed under surveillance during the execution of the program. Each time an identifier included in the monitor list is used in one of the ways described below, the identifier and its current value (unless it is a label) are written on the file indicated in the monitor declaration.

When a simple variable in the monitor list is used as a left part in an assignment statement, the following information is written on the designated file:

⟨simple variable⟩ = {value of variable}

When a subscripted variable in the monitor list is encountered during the execution of the program as the leftmost element in a left part list, the following information is written on the designated file:

⟨array identifier⟩[{value of subscript expression}]=
{value of variable}

When a subscripted variable, the array identifier of which is given in the monitor list, is encountered as the leftmost element in a left part list, the following information is written on the designated file:

⟨array identifier⟩[{value of subscript expression}]=
{value of variable}

Each time a statement bearing a label which is in the monitor list is encountered in the program, the label identifier is written on the designated file.

A monitor declaration might appear in a program in the following manner.

```
          BEGIN                                                AA000001
FILE ØUT        LINE 1(2,15);                                  AA000002
ARRAY           X, Y[0:10,0:20] , T [0:150 ],                  AA000003
                   KX [0:10];                                  AA000004
REAL            MAX, P, Q;                                     AA000005
INTEGER         A, K, I;                                       AA000006
LABEL           LØ, HI, XIT, REED;                             AA000007
MØNITØR         LINE (X, Y, MAX, HI, I, LØ,                    AA000008
                  T[K]);                                       AA000009
```

Here, diagnostic output would occur after the execution of statements, such as:

```
X[I,K] ← Y [I,K] × P - Q;

MAX ← T [0] /10;

HI:        K ← I;

I ← I + 2;

Y [0,0 ] ← KX [0,0] ← MAX;

T [K] ← 40;
```

No monitor action would occur for statements, such as:

```
FOR I ← 1,2,4 DO KX [I] ← 1/KX [I];

REED: READ (CARD, FIN, MAX);

K ← I;

KX [0,0] ← Y [0,0] ← MAX;

T [I] ← 40;
```

If it were desired to monitor values of MAX obtained with the read statement, the statement MAX ← MAX could be added. The dump declaration has the form

DUMP ⟨dump part⟩,

where the latter portion is formed according to the rules

⟨dump part⟩ ::= ⟨file identifier⟩ (⟨dump list⟩) ⟨label⟩: ⟨dump indicator⟩

⟨dump list⟩ ::= ⟨dump list element⟩ | ⟨dump list⟩, ⟨dump list element⟩

⟨dump list element⟩ ::= ⟨simple variable⟩ | ⟨subscripted variable⟩ | ⟨label⟩ | ⟨array identifier⟩

⟨dump indicator⟩ ::= ⟨unsigned integer⟩ | ⟨simple variable⟩

Diagnostic information requested by means of the dump declaration is written on the designated file when a statement, carrying the label which is given in the dump part, has been executed a number of times equal to the dump indicator. Since the dump indicator can be a simple variable, dump information can be obtained more than once during each execution of the block containing the dump declaration. The number of times the controlling statement is executed

applies to each pass through that block. The number is not cumulative from one pass to the next.

The actions obtained for each kind of dump list element are described below.

A simple variable in the dump list causes the current value of that variable to be supplied in the following form:

⟨simple variable⟩ = {value of variable}

A subscripted variable in the dump list causes the current value of that variable to be supplied in the following form:

⟨array identifier⟩[{value of subscript expression}]=
{value of variable}

An array identifier in the dump list causes the current values of all elements in that array to be supplied in tabular form and identified by

⟨array identifier⟩ =

which appears just above the table.

The order in which the array elements are written is as follows: All subscripts are first set to their declared lower bounds, and the corresponding value is printed out. The rightmost subscript is then counted up, and the corresponding value is printed; this procedure continues until the subscript reaches its declared upper bound. After this printout, the rightmost subscript is again set to its declared lower bound, the next left subscript is counted up, and the process recycles until all subscripts have reached their declared upper bounds.

A label in the dump list causes a tally to be supplied which represents the number of times the labeled statement has been executed during this pass through the block containing the dump declaration. The tally is supplied in the following form:

⟨label⟩ {number of times statement has been executed}

A dump declaration that might be used in the previous example is
DUMP LINE (HI, KX, Y, MAX, REED) XIT: 1

Finally, attention is called to a restriction on the use of any input-output, including monitor and dump, in the body of typed procedures which are called via function designators in expressions occurring in the list of a write statement. All input-output editing is done by a set of intrinsic procedures, which are not capable of intertwined processing of an output list along with one or more other input or output lists.

Appendix A - RESERVED WORDS

The following list includes all of the normally reserved words of B 5500 Extended ALGOL. They must be respected as such, even though they may not occur in the subset presented in this text.

ABS	FALSE	OUT	VALUE
ALPHA	FILE	OWN	WHILE
AND	FILL	PAGE	WITH
ARCTAN	FOR	PROCEDURE	WRITE
ARRAY	FORMAT	READ	
BEGIN	FORWARD	REAL	
BOOLEAN	GO	RELEASE	
CLOSE	IF	REVERSE	
COMMENT	IMP	REWIND	
COS	IN	SAVE	
DBL	INTEGER	SIGN	
DEFINE	LABEL	SIN	
DIV	LIST	SPACE	
DO	LN	SQRT	
DOUBLE	LOCK	STEP	
DUMP	MOD	STREAM	
ELSE	MONITOR	SWITCH	
END	NO	THEN	
ENTIER	NOT	TO	
EQV	OCT	TRUE	
EXP	OR	UNTIL	

This appendix is presented in two parts:

Part I contains the entire syntax in logical order. Part II contains a list of all metalinguistic variables in alphabetical order.

Appendix B - SYNTAX

This appendix is presented in two parts:

Part I contains the entire syntax in logical order. Part II contains a list of all metalinguistic variables in alphabetical order.

PART I

SYNTAX IN LOGICAL ORDER

The number appearing on the left is a formula number which is referred to by Part II.

1. ⟨basic symbol⟩ ::= ⟨letter⟩ | ⟨digit⟩ | ⟨delimiter⟩

2. ⟨letter⟩ ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

3. ⟨digit⟩ ::= 0|1|2|3|4|5|6|7|8|9

4. ⟨delimiter⟩ ::= ⟨separator⟩ | ⟨bracket⟩ | ⟨declarator⟩ | ⟨arithmetic operator⟩ | ⟨relational operator⟩ | ⟨sequential operator⟩ | ⟨logical operator⟩

5. ⟨arithmetic operator⟩ ::= +|-|×|/|*

6. ⟨relational operator⟩ ::= ≤|<|=|≥|>|≠

7. ⟨logical operator⟩ ::= OR|AND

8. ⟨sequential operator⟩ ::= GO|TO|IF|THEN|ELSE|FOR|DO|READ|WRITE|PAGE

9. ⟨separator⟩ ::= ,|.|@|:|;|←|⟨single space⟩|STEP|UNTIL|COMMENT

10. ⟨single space⟩ ::= {a single unit of horizontal spacing which is blank}

11. ⟨bracket⟩ ::= (|)|[|]|"|BEGIN|END

12. ⟨declarator⟩ ::= INTEGER|REAL|ARRAY|LABEL|LIST|FORMAT|IN|OUT|MONITOR|DUMP|FILE|PROCEDURE

13. ⟨identifier⟩ ::= ⟨letter⟩ | ⟨identifier⟩ ⟨letter⟩ | ⟨identifier⟩⟨digit⟩

14. ⟨number⟩ ::= ⟨unsigned number⟩ | +⟨unsigned number⟩ | -⟨unsigned number⟩

15. ⟨unsigned number⟩ ::= ⟨decimal number⟩ | ⟨exponent part⟩ | ⟨decimal number⟩ ⟨exponent part⟩

16. ⟨decimal number⟩ ::= ⟨unsigned integer⟩ | ⟨decimal fraction⟩ | ⟨unsigned integer⟩ ⟨decimal fraction⟩

17. ⟨exponent part⟩ ::= @⟨integer⟩

18. ⟨decimal fraction⟩ ::= .⟨unsigned integer⟩

19. ⟨integer⟩ ::= ⟨unsigned integer⟩ | + ⟨unsigned integer⟩ | -⟨unsigned integer⟩

20. ⟨unsigned integer⟩ ::= ⟨digit⟩ | ⟨unsigned integer⟩ ⟨digit⟩

21. ⟨variable⟩ ::= ⟨simple variable⟩ | ⟨subscripted variable⟩

22. ⟨simple variable⟩ ::= ⟨identifier⟩

23. ⟨type declaration⟩ ::= ⟨type⟩ ⟨type list⟩

24. ⟨type⟩ ::= REAL | INTEGER

25. ⟨type list⟩ ::= ⟨simple variable⟩ | ⟨type list⟩ , ⟨simple variable⟩

26. ⟨primary⟩ ::= ⟨unsigned number⟩ | ⟨variable⟩ |
⟨function designator⟩ | (⟨arithmetic expression⟩)

27. ⟨function designator⟩ ::= ⟨common function identifier⟩
(⟨arithmetic expression⟩) |
⟨procedure identifier⟩
⟨actual parameter part⟩

28. ⟨common function identifier⟩ ::= SQRT | SIN | COS | ARCTAN | LN |
EXP | ABS | SIGN | ENTIER

29. ⟨adding operator⟩ ::= + | -

30. ⟨multiplying operator⟩ ::= × | /

31. ⟨arithmetic expression⟩ ::= ⟨term⟩ | ⟨adding operator⟩
⟨term⟩ | ⟨arithmetic expression⟩
⟨adding operator⟩ ⟨term⟩

32. ⟨term⟩ ::= ⟨factor⟩ | ⟨term⟩ ⟨multiplying operator⟩ ⟨factor⟩

33. ⟨factor⟩ ::= ⟨primary⟩ | ⟨factor⟩ * ⟨primary⟩

34. ⟨statement⟩ ::= ⟨unconditional statement⟩ |
⟨conditional statement⟩ | ⟨for statement⟩

35. ⟨declaration⟩ ::= ⟨type declaration⟩ | ⟨array declaration⟩ |
⟨label declaration⟩ | ⟨file declaration⟩ |
⟨format declaration⟩ | ⟨list declaration⟩ |
⟨diagnostic declaration⟩ |
⟨procedure declaration⟩

36. ⟨unconditional statement⟩ ::= ⟨compound statement⟩ |
⟨block⟩ | ⟨basic statement⟩

37. ⟨basic statement⟩ ::= ⟨unlabelled basic statement⟩ |
⟨label⟩ : ⟨basic statement⟩

38. ⟨unlabelled basic statement⟩ ::= ⟨assignment statement⟩ |
⟨go to statement⟩ |
⟨dummy statement⟩ |
⟨read statement⟩ |
⟨write statement⟩ |
⟨procedure statement⟩

39. ⟨compound statement⟩ ::= ⟨unlabelled compound statement⟩ |
⟨label⟩ : ⟨compound statement⟩

40. ⟨unlabelled compound statement⟩ ::= BEGIN ⟨compound tail⟩

41. ⟨compound tail⟩ ::= ⟨statement⟩ END | ⟨statement⟩ ;
⟨compound tail⟩

42. ⟨block⟩ ::= ⟨unlabelled block⟩ | ⟨label⟩ : ⟨block⟩

43. ⟨unlabelled block⟩ ::= ⟨block head⟩ ; ⟨compound tail⟩

44. ⟨block head⟩ ::= BEGIN ⟨declaration⟩ | ⟨block head⟩ ;
⟨declaration⟩

45. ⟨program⟩ ::= ⟨unlabelled block⟩ . |
⟨unlabelled compound statement⟩.

46. ⟨assignment statement⟩ ::= ⟨left part list⟩ ⟨arithmetic expression⟩

47. ⟨left part list⟩ ::= ⟨left part⟩ | ⟨left part list⟩ ⟨left part⟩

48. ⟨left part⟩ ::= ⟨variable⟩←

49. ⟨go to statement⟩ ::= GO TO ⟨label⟩

50. ⟨label declaration⟩ ::= LABEL ⟨label list⟩

51. ⟨label list⟩ ::= ⟨label⟩ | ⟨label list⟩ , ⟨label⟩

52. ⟨label⟩ ::= ⟨identifier⟩

53. ⟨dummy statement⟩ ::= ⟨empty⟩

54. ⟨empty⟩ ::={the null string of symbols}

55. ⟨Boolean primary⟩ ::= ⟨relations⟩ | (⟨Boolean expression⟩)

56. ⟨relation⟩ ::= ⟨arithmetic expression⟩ ⟨relational operator⟩
⟨arithmetic expression⟩

57. ⟨Boolean expression⟩ ::= ⟨Boolean factor⟩ | ⟨Boolean expression⟩
OR ⟨Boolean factor⟩

58. ⟨Boolean factor⟩ ::= ⟨Boolean primary⟩ | ⟨Boolean factor⟩ AND
⟨Boolean primary⟩

59. ⟨if statement⟩ ::= ⟨if clause⟩ ⟨unconditional statement⟩

60. ⟨if clause⟩ ::= IF ⟨Boolean expression⟩ THEN

61. ⟨conditional statement⟩ ::= ⟨if statement⟩ | ⟨if statement⟩ ELSE ⟨statement⟩ | ⟨if clause⟩ ⟨for statement⟩ | ⟨label⟩: ⟨conditional statement⟩

62. ⟨for statement⟩ ::= ⟨for clause⟩ ⟨statement⟩ | ⟨label⟩: ⟨for statement⟩

63. ⟨for clause⟩ ::= FOR ⟨controlled variable⟩ ← ⟨for list⟩ DO

64. ⟨controlled variable⟩ ::= ⟨simple variable⟩

65. ⟨for list⟩ ::= ⟨for list element⟩ | ⟨for list⟩ , ⟨for list element⟩

66. ⟨for list element⟩ ::= ⟨arithmetic expression⟩ | ⟨arithmetic expression⟩ STEP ⟨arithmetic expression⟩ UNTIL ⟨arithmetic expression⟩

67. ⟨subscripted variable⟩ ::= ⟨array identifier⟩ [⟨subscript list⟩]

68. ⟨array identifier⟩ ::= ⟨identifier⟩

69. ⟨subscript list⟩ ::= ⟨subscript expression⟩ | ⟨subscript list⟩ , ⟨subscript expression⟩

70. ⟨subscript expression⟩ ::= ⟨arithmetic expression⟩

71. ⟨array declaration⟩ ::= ⟨array kind⟩ ARRAY ⟨array list⟩

72. ⟨array kind⟩ ::= ⟨empty⟩ | ⟨type⟩

73. ⟨array list⟩ ::= ⟨array segment⟩ | ⟨array list⟩ , ⟨array segment⟩

74. ⟨array segment⟩ ::= ⟨array identifier⟩ [⟨bound pair list⟩] | ⟨array identifier⟩ , ⟨array segment⟩

75. ⟨bound pair list⟩ ::= ⟨bound pair⟩ | ⟨bound pair list⟩ , ⟨bound pair⟩

76. ⟨bound pair⟩ ::= ⟨lower bound⟩ : ⟨upper bound⟩

77. ⟨lower bound⟩ ::= ⟨integer⟩

78. ⟨upper bound⟩ ::= ⟨integer⟩

79. ⟨file declaration⟩::= FILE ⟨I-0 part⟩ ⟨file part⟩

80. ⟨I-0 part⟩ ::= IN|OUT

81. ⟨file part⟩ ::= ⟨file identifier⟩ ⟨unit designation⟩
(⟨buffer part⟩)

82. ⟨file identifier⟩ ::= ⟨identifier⟩

83. ⟨unit designation⟩ ::= ⟨empty⟩|⟨space⟩1

84. ⟨space⟩ ::= ⟨single space⟩ | ⟨space⟩ ⟨single space⟩

85. ⟨buffer part⟩ ::= ⟨number of buffers⟩ , ⟨buffer size⟩

86. ⟨number of buffers⟩ ::= ⟨unsigned integer⟩

87. ⟨buffer size⟩ ::= ⟨unsigned integer⟩

88. ⟨format declaration⟩ ::= FORMAT ⟨I-0 part⟩ ⟨format part⟩|
FORMAT ⟨format part⟩

89. ⟨format part⟩ ::= ⟨format identifier⟩(⟨editing specifications⟩)|
⟨format part⟩ , ⟨format identifier⟩
(⟨editing specifications⟩)

90. ⟨format identifier⟩ ::= ⟨identifier⟩

91. ⟨editing specifications⟩ ::= ⟨editing segment⟩|
⟨editing specifications⟩/|
/⟨editing specifications⟩|
⟨editing specifications⟩/
⟨editing segment⟩

92. ⟨editing segment⟩ ::= ⟨editing phrase⟩ | ⟨repeat part⟩
(⟨editing specifications⟩) | ⟨editing segment⟩,
⟨editing phrase⟩ | ⟨editing segment⟩,
⟨repeat part⟩ (⟨editing specifications⟩)

93. ⟨editing phrase⟩ ::= ⟨repeat part⟩ ⟨editing phrase type⟩
⟨field part⟩ | ⟨string⟩

94. ⟨repeat part⟩ ::= ⟨empty⟩ | ⟨unsigned integer⟩

95. ⟨editing phrase type⟩ ::= E|F|R|I|X

96. ⟨field part⟩ ::= ⟨field width⟩ | ⟨field width⟩. ⟨decimal places⟩

97. ⟨field width⟩ ::= ⟨unsigned integer⟩

98. ⟨decimal places⟩ ::= ⟨unsigned integer⟩

99. ⟨string⟩ ::= "{ any string of valid characters except"} "

100. ⟨list declaration⟩ ::= LIST ⟨list part⟩

101. ⟨list part⟩ ::= ⟨list identifier⟩(⟨list⟩) | ⟨list part⟩,
⟨list identifier⟩(⟨list⟩)

102. ⟨list identifier⟩ ::= ⟨identifier⟩

103. ⟨list⟩ ::= ⟨list segment⟩ | ⟨list⟩, ⟨list segment⟩

104. ⟨list segment⟩ ::= ⟨expression part⟩ | ⟨for clause⟩
⟨list segment⟩ | ⟨for clause⟩
[⟨expression list⟩]

105. ⟨expression part⟩ ::= ⟨arithmetic expression⟩

106. ⟨expression list⟩ ::= ⟨expression part⟩ | ⟨expression list⟩,
⟨expression part⟩ | ⟨list segment⟩ |
⟨expression list⟩, ⟨list segment⟩

107. ⟨read statement⟩ ::= READ (⟨input parameters⟩)
⟨action label⟩

108. ⟨input parameters⟩ ::= ⟨file identifier⟩, ⟨format identifier⟩,
⟨list⟩ | ⟨file identifier⟩,
⟨format identifier⟩, ⟨list identifier⟩

109. ⟨action label⟩ ::= [⟨end of file label⟩] | ⟨empty⟩

110. ⟨end of file label⟩ ::= ⟨label⟩

111. ⟨write statement⟩ ::= WRITE (⟨output parameters⟩)

112. ⟨output parameters⟩ ::= ⟨file identifier⟩ ⟨format and list parameters⟩

113. ⟨format and list parameters⟩ ::= ⟨format and list part⟩ |
⟨empty⟩ | [⟨skip to next page⟩]

114. ⟨format and list part⟩ ::= , ⟨format identifier⟩ |
, ⟨format identifier⟩, ⟨list⟩ |
, ⟨format identifier⟩
, ⟨list identifier⟩

115. ⟨skip to next page⟩ ::= PAGE

116. ⟨procedure declaration⟩ ::= PROCEDURE ⟨procedure heading⟩
⟨procedure body⟩ | ⟨type⟩ PROCEDURE
⟨procedure heading⟩ ⟨procedure body⟩

117. ⟨procedure body⟩ : : = ⟨statement⟩

118. ⟨procedure heading⟩ : : = ⟨procedure identifier⟩
⟨formal parameter⟩ ;
⟨specification part⟩

119. ⟨procedure identifier⟩ : : = ⟨identifier⟩

120. ⟨formal parameter part⟩ : : = ⟨empty⟩ | (⟨identifier list⟩)

121. ⟨identifier list⟩ : : = ⟨identifier⟩ | ⟨identifier list⟩ , ⟨identifier⟩

122. ⟨specification part⟩ : : = ⟨empty⟩ | ⟨specification list⟩

123. ⟨specification list⟩ : : = ⟨specification⟩ ; | ⟨specification list⟩
⟨specification⟩

124. ⟨specification⟩ : : = ⟨specifier⟩ ⟨identifier list⟩ |
⟨array specification⟩

125. ⟨specifier⟩ : : = ⟨type⟩ | FILE | LIST | FORMAT | LABEL

126. ⟨array specification⟩ : : = ARRAY ⟨array specifier list⟩ | ⟨type⟩
ARRAY ⟨array specifier list⟩

127. ⟨array specifier list⟩ : : = ⟨array specifier⟩ |
⟨array specifier list⟩,
⟨array specifier⟩

128. ⟨array specifier⟩ : : = ⟨array identifier list⟩
[⟨lower bound list⟩]

129. ⟨array identifier list⟩ : : = ⟨identifier list⟩

130. ⟨lower bound list⟩ : : = ⟨specified lower bound⟩ |
⟨lower bound list⟩,
⟨specified lower bound⟩

131. ⟨specified lower bound⟩ : : = ⟨integer⟩ | *

132. ⟨procedure statement⟩ : : = ⟨procedure identifier⟩
⟨actual parameter part⟩

133. ⟨actual parameter part⟩ : : = ⟨empty⟩ | (⟨actual parameter list⟩)

134. ⟨actual parameter list⟩ : : = ⟨actual parameter⟩ |
⟨actual parameter list⟩,
⟨actual parameter⟩

135. ⟨actual parameter⟩ : : = ⟨arithmetic expression⟩ | ⟨array identifier⟩ |
⟨file identifier⟩ | ⟨format identifier⟩ |
⟨list identifier⟩ | ⟨label⟩

136. ⟨diagnostic declaration⟩ ::= MONITOR ⟨monitor part⟩ |
DUMP ⟨dump part⟩

137. ⟨monitor part⟩ ::= ⟨file identifier⟩ (⟨monitor list⟩)

138. ⟨monitor list⟩ ::= ⟨monitor list element⟩ | ⟨monitor list⟩ ,
⟨monitor list element⟩

139. ⟨monitor list element⟩ ::= ⟨simple variable⟩ |
⟨subscripted variable⟩ |
⟨array identifier⟩ | ⟨label⟩

140. ⟨dump part⟩ ::= ⟨file identifier⟩ (⟨dump list⟩) ⟨label⟩ :
⟨dump indicator⟩

141. ⟨dump list⟩ ::= ⟨dump list element⟩ | ⟨dump list⟩ ,
⟨dump list element⟩

142. ⟨dump list element⟩ ::= ⟨simple variable⟩ | ⟨subscripted variable⟩ |
⟨label⟩ | ⟨array identifier⟩

143. ⟨dump indicator⟩ ::= ⟨unsigned integer⟩ | ⟨simple variable⟩

PART II

METALINGUISTIC VARIABLES IN ALPHABETICAL ORDER

Two columns of numbers appear to the right of each variable. The first column of numbers references, in Part I, the formula given for this variable. The second column of numbers refers to a paragraph number where the formula is discussed in the text.

Appendix C - DECLARATIONS

Declarations provide the compiler with required information about identifiers used in a program. All identifiers must be explicitly defined by a declaration. The following lists the eight declarations and briefly points out the purpose of each. The reference following each definition indicates the section in the text where the declaration is discussed.

⟨type declaration⟩ - Specifies whether a simple variable is REAL or INTEGER (2.5).

⟨array declaration⟩ - Specifies that an identifier represents an array and gives the number of dimensions in the array and the bounds on each dimension (6.2).

⟨label declaration⟩ - Specifies that certain identifiers are labels (3.5).

⟨file declaration⟩ - Associates an identifier with a set of file-handling specifications (7.1).

⟨format declaration⟩ - Associates an identifier with a set of editing specifications (7.2).

⟨list declaration⟩ - Associates one identifier with a set of expressions to be initialized (in the case of input) or printed (in the case of output) (7.3).

⟨procedure declaration⟩ - Associates an identifier with a body of coding which is to be treated as a subprogram (8.2) (9.3).

⟨diagnostic declaration⟩ - Associates an identifier with either monitoring or dumping specifications (10.4).

Appendix D - DEVIATIONS FROM ALGOL 60

Reference was made in Chapter 1 to ALGOL 60, the language on which B 5500 ALGOL is based. Although the latter contains many extensions to the language, not many of these have been included in the subset presented in this text. Therefore, with few exceptions, this subset is also a subset of ALGOL 60. This text can thus serve well as an introduction to ALGOL 60.

A student, who has learned the material in this text, can readily pinpoint the few differences by consulting the ALGOL reference publications and other discussions of ALGOL*.

Except for Chapters 7 and 10, most of the concepts presented here have exact counterparts in ALGOL 60. The important areas of difference are discussed below.

Input-Output Constructs - No facilities for communication of data and results are specified in ALGOL 60.

Labels - In ALGOL 60, it is permissible to use unsigned integers, as well as identifiers, for labels. No label declaration exists in ALGOL 60.

Character Set - The B 5500 character set does not include all of the characters used in ALGOL 60. As a substitute, reserved words such as AND and OR are used. The replacement operator in ALGOL 60 is the character pair : = whereas the B 5500 ALGOL compiler accepts either ← or := as a replacement operator. ALGOL 60 uses the symbol ↑ for exponentiation, rather than the asterisk. Powers of ten in numbers are written with an undersized 10 in ALGOL 60 while the @ symbol is used on the B 5500.

Reserved Words - There are no reserved words in ALGOL 60. Delimiters such as BEGIN, STEP and IF are considered to be single symbols, (see Section 2.2). The blank is not used as a separator in ALGOL 60, whereas in the B 5500 it is often needed in conjunction with reserved words, (see Section 2.3).

Identifier Length - No limit on identifier length is given in ALGOL 60.

Specifications - The specifications, which are a part of procedure headings that contain formal parameters, are not mandatory in ALGOL 60. In an array specification, no lower bound list is given in ALGOL 60.

GO TO's in Procedure Bodies - No restriction on the use of a go to statement referencing a label nonlocal to the body of a typed procedure is given in ALGOL 60, (see Section 9.3).

Global Identifiers in Procedure Bodies - The restriction discussed in Section 8.2 is not present in ALGOL 60.

*"Structure and Use of ALGOL 60," Bottenbruch, H.,
Journal of the Association for Computing Machinery, April, 1962

"An Introduction to ALGOL 60," Schwarz, H. R.,
Communications of the Association for Computing Machinery, February, 1962

1019973
(Formerly 5000-21030)

Printed in U. S. America