AN INTRODUCTORY COURSE IN
COMPUTER PROGRAMMING

Course Material Developed for the Discrete System Concepts Project.

Robert T. Braden
Alan J. Perlis
Computation Center
Carnegie Institute of Technology

M o n o g r a p h No. 7

This is a publication of the Discrete System Concepts Project--funded by the National Science Foundation under Grant No. GE-2546, involving:

Carnegie Institute of Technology
Case Institute of Technology
Illinois Institute of Technology
Polytechnic Institute of Brooklyn
Rensselaer Polytechnic Institute

15 June 1965

F O R E W O R D

The course described herein is the means by which a university student is introduced to complex computation. This first contact should happen at the earliest possible time in his college education, and the contact should be analytical, not descriptive. Together with courses in mathematics and natural language, this course should contribute to his development of fluency in the use of intellectual tools.

It is hoped that the material presented here will be helpful in organizing a first one-semester course in computing. Because this course is limited to one-semester, it does not contain many things that a programming course should. Succeeding courses involve the student in symbol manipulation, <u>per se</u>, and in the details of designing real and, hence, complex computer systems.

The operation of this course is supported by a digital computer and its associated programming systems. Since the intent is to make the computer a natural tool for the engineer and scientist, the student should be allowed to make use of the computer in his other technical courses during succeeding semesters.

A few comments about the operation of the computers at Carnegie Tech might be in order. The current system is a Control Data Corporation G-21 possessing two processors,a large core storage ($\sim$ 65,000 words) and disc files having 48,000,000 characters of storage.

While everyone must compose and prepare his own program, the computer processing of programs is accomplished by a professional staff. Thus, experience in equipment tending is <u>not</u> part of the education arising from this course.

The student is treated in no way differently from faculty or staff in the nature of, duration of, or frequency of his contact with the processing capacity of the computer. One of the by-products of the course is familiarity with the administrative system that inevitably surrounds access to every large computer system -- familiarity with the many computer languages, library routines, input teletypes, etc.

This course is an introduction to the fledgling subject of computer science and should be followed by access to baccalaureate (and doctoral) programs in this area. Carnegie provides this access through options in the mathematics and electrical engineering programs. Both are described in the school catalogue.

C O N T E N T S

PART I - INTRODUCTION

The time is approaching when all undergraduate majors in engineering and the physical sciences will be expected to receive at least an introductory course in computer programming. This is simply a reflection of the increasingly important role of computing machines in the professional work of these fields. For example, the new interest in the discrete system approach to engineering problems is due in considerable measure to the availability of modern high-speed computing machines to perform the extensive numerical computations which these methods frequently entail. Hence a programming course must form an essential part of a curriculum including discrete system concepts.

A course in computer programming can be taught from any one of many different points of view, depending upon its function in the curriculum. Thus, a course in any of the following three areas could be considered a "course in introductory computer programming:"

(1) Applied Numerical Analysis.
This would be partly or entirely a mathematics course, perhaps included in the standard n-semester sequence of analytic geometry, calculus, and differential equations. Such a course might teach just enough programming to allow students to use a computer to solve problems in approximation, numerical quadrature, numerical solution of differential equations, etc.

(2) Problem Solving.
Such a programming course would consist almost entirely of a set of problems whose solution required the student to employ analytic and creative thinking to develop a suitable algorithm. This course would attempt to develop general problem-solving skills in the student, with computer programming serving only as a source of problems to be solved. The computer itself would sit in final judgment, determining quite objectively the success of the student's intellectual effort.

(3) Theory and Techniques of Programming.
This course would focus on programming as a distinct field of knowledge with its own concepts and techniques (without, however, losing sight of the idea of a computer as a computational tool). The course would explore methods of organizing data and programs, of defining computer languages, and of dealing with the severe limitations of machine time and memory space which plague "real" programming problems.

Most introductory programming courses will contain some mixture of these three components. This monograph describes a course* developed to emphasize the last two components: problem solving, and programming theory and techniques.

The central task in such an introductory programming course is to teach the students a working knowledge of an algorithmic programming language; we have chosen to use the language ALGOL-60, for reasons discussed later. Such a "higher level" language has several essential pedagogic advantages over machine language for an introductory course. Using a language like ALGOL, the students can understand and create relatively complex algorithms and programs and can solve meaningful problems almost immediately on the computer; furthermore, the instructor can explain important concepts of data and program organization -- for example: arrays, mapping functions, name and value parameters, and dynamic storage allocation -- much more easily. Machine language should enter such a course only in simplified, "cleaned-up" form, as background for understanding the meaning and implementation of ALGOL constructs. For those students who wish to study programming further, there should be advanced courses covering machine language as well as compiler construction, monitor systems, and list processing languages.

The course which we are describing would be organized with class meetings divided between formal lectures and small discussion sections. A suggested outline for the lectures appears in Part II; in general, they should cover introductory material on programming and computation, including the following topics: algorithms and their pitfalls, flow charts, ALGOL, data structures, program organization, computer organization and machine language, and the precise specification of mechanical languages. The discussion sections, containing at most 20 to 25 students each, would be largely devoted to explanation and discussion of the specifics of ALGOL programming.

A suitable basic textbook for the programming course would be "A Guide to ALGOL Programming", by Daniel D. McCracken; this and other suitable books are listed in the Bibliography, Part V. McCracken's text is a clear introduction to ALGOL, and contains a good set of examples drawn from the fields of engineering and physical science. However, we feel that it is desirable to supplement this text, going more deeply into areas including the following:

---

* The prototype of the course described in this monograph is a course (S205) taught for the past five years at Carnegie Institute of Technology.

*1. Flow charts and algorithms.

2. The grammatical structure of an ALGOL program; for example: explanation of the punctuation rules in terms of the syntactic structure of statements, compound statements, etc.; conditional expressions; and the "dangling else" ambiguity in ALGOL.

*3. Data structures: arrays, mapping functions, access tables, and trees.

*4. Procedures and subroutines.

5. Block structure and dynamic array storage allocation.

6. Machine language and, in particular, the machine language representation of an ALGOL program.

*7. Backus Normal Form for definition of programming language syntax.

Supplementary notes have been prepared to cover the topics in this list which are prefixed with an asterisk; these notes are contained in Part III of this report, together with some sample homework and examination problems related to each topic.

Programming is a skill which is best taught with a balanced combination of (1) lectures and text book readings, and (2) experience gained through solving programming problems. Therefore, it is of fundamental importance that students in an introductory course write ALGOL programs to solve 5 to 15 practice problems and debug their programs on a computer. In general, these problems can be divided into two classes: (1) programming technique problems, which in conjunction with the lectures are intended to teach the student particular concepts; and (2) algorithmic problems, which are exercises in analyzing the structure of problems and in developing suitable algorithms for their solution. It is the latter problems which provide the "problem solving" component of the course. A sample of suitable problems, ranging from very easy to very difficult, is contained in Part IV.

The students are provided with library procedures, called "TEACH" procedures, to aid them in debugging their programs on the computer. A TEACH procedure supplies test data to their program and checks their results; the same TEACH procedure is also used to mechanically grade the problem solutions. The use of TEACH procedures is explained in Part IV.

We should comment upon the choice of the language ALGOL for teaching an introductory programming course. In fact, the students must learn two dialects of ALGOL: the official language ALGOL-60 as well as the particular hardware representation which is available to them on the computer. The differences between these dialects should be minor; if so, the instructor should avoid emphasizing the distinction but instead treat the two languages as members of a family of nearly identical languages, all simply "ALGOL". For example, ALGOL-20* allows the symbol "$\leftarrow$" to be used interchangeably with the ":=" symbol of ALGOL-60 as an assignment operator. In this monograph, "$\leftarrow$" has been used in preference to ":=" even when the text is nominally the Reference Language ALGOL-60. It has been observed over four semesters that students readily accept this mild linguistic schizophrenia.

Since ALGOL was designed as a publication language and is therefore quite readable, it is useful for describing algorithms and programming concepts even if a completely different language is taught to the students for actual programming. Before an ALGOL translator was available at Carnegie Tech, the introductory programming course taught a locally-developed FORTRAN-like language called GATE; even at that time ALGOL was introduced in the lectures as a natural and efficient way of stating algorithms. Thus, it was much easier to write:

$$\underline{\text{for}}\ I \leftarrow 1\ \underline{\text{step}}\ 1\ \underline{\text{until}}\ 20\ \underline{\text{do}}\ A_{I-1} \leftarrow A_{I-2} + 4\ ;$$

or

$$Y \leftarrow (\ \underline{\text{if}}\ X > Y\ \underline{\text{then}}\ X\ \underline{\text{else}}\ Y\ )$$

than to draw the corresponding flowcharts on a blackboard; very little explanation was needed to give the students a reading knowledge sufficient to interpret these statements.

Several other advantages of ALGOL for an introductory course can be cited. ALGOL is an international standard language for publishing algorithms; many important algorithms have been and will be published in this form. Hence it is useful for a student to have at least a reading knowledge of the language. The ALGOL language has a simple and logically clean structure, but is a rich language; furthermore, it has established a standard nomenclature which is very useful for talking about computer programming and other computer languages.

* "ALGOL-20" is the hardware representation of ALGOL-60 developed at Carnegie Institute of Technology for the CDC G-20 computer.

PART II - AN OUTLINE OF LECTURES FOR AN INTRODUCTORY PROGRAMMING COURSE

A. Algorithms. (4 lectures)

1. Necessity of algorithms for numerical computation.
2. Algorithmic languages: flow charts, ALGOL.
   - a. Flow charts.
     - (1). Execution of a flow chart.
     - (2). Flow charts within flow charts: hierarchy, down to "primitive" flow charts which use only the "basic" arithmetic and test operations.
     - (3). Examples of simple flow charts; loops.
   - b. ALGOL.
     - (1). The dual role of ALGOL -- communicating algorithms and programming a computer.
     - (2). ALGOL program: sequence of statements (commands).
     - (3). Order of execution; rules for determining successor. GO TO and IF statements. Compound statements.
     - (4). ALGOL syntactic structure: Identifiers, expressions, statements, declarations.

B. Programs and Computers. (2 lectures)

1. Organization of computer: memory, arithmetic unit, and control unit.
2. Fetch/Execute cycle. Sequential execution.
3. Relationship of ALGOL names to computer memory. Addresses.
4. Translation vs. execution of ALGOL program. Existence of a "machine language representation" of the source program.

C. Data structures and representations. (3-4 lectures)

1. Arrays, the primary data structure of ALGOL.
   - a. ALGOL subscripted variables:
     Declarations, subscript bounds, dimensions.

   b. Examples of the use of arrays in ALGOL programs.
      Representations of tables, complex numbers, polynomials.

   c. The memory mapping functions for ALGOL arrays.
      Access tables.

2. More general data structure.

   a. General access tables, "jagged" arrays.

   b. Trees.

   c. The problem of variable length information: compact storage, insertion, and deletion; list structures.

Program Structures: Subroutines, Procedures and Blocks. (3-4 lectures)

1. Subroutine: Segment of program which can be executed as a single "statement" (or instruction) in a "master" program.

   a. Subroutine linkage.

   b. ALGOL: Declaration vs. call of procedure.

2. Generalization of procedures by parametrization.

   a. Formal, actual parameters.

   b. Name, value parameters.

   c. Calculated call-by-name.

3. Subroutines for program organization.

   a. Definition of new (non-primitive) "elementary processes."

   b. Independent subprograms: "Black Boxes", local variables.
      Publication of algorithms as ALGOL procedures.

   c. Examples of the use of procedures for complex program organization.

   d. Recursive procedures.

4. ALGOL Block structure.

   a. Local vs. global names.

   b. Level of nesting.

   c. Dynamic storage allocation; block administration.

E. Computing machine structure and operation. (2-3 lectures)

   1. Number representations, and conversion of radix.

   2. Elementary machine language.

   3. Machine language representation of ALGOL programs.

4. The concepts of immediate, normal, and indirect addressing.

5. Machine language meaning of subroutines, call-by-name, call-by-value, and array accessing.

F. Mechanical Languages. (1-2 lectures)

1. Meta- vs. object-languages.

2. Backus Normal Form.

PART III - SUPPLEMENTARY LECTURE MATERIAL

1. ALGORITHMS AND FLOWCHARTS

A. The Nature of a Computer

In order to understand computer programming, one must understand clearly the basic organization and nature of a computer. As used in this course, the term "computer" is really short for the complete expression: "general purpose automatic digital computer". We could have further lengthened this title by inserting the adjective "electronic"; all modern digital computers do operate with electronic circuitry although in the past they have been built of mechanical and electromagnetic parts and in the future they might be pneumatic, hydraulic, or optical. In any case, they could be made of Erector Sets or green cheese as far as we are concerned here; we are only interested in how they are organized and instructed to perform calculations.

A digital computer is a machine which performs numerical computations -- that is, ordinary arithmetic -- on numbers represented in a discrete (or "digital") notation; thus, a digital computer is a descendant of a mechanical adding machine, each of whose dials has ten discrete positions, rather than of a slide rule which is a continuous (or "analog") computer. A digital computer has a particular set of elementary operations wired in; we will refer to these operations as "primitive". Most computers include the following primitive operations:

1. The normal arithmetic operations: addition, subtraction, multi-

plication and division, symbolized by:

$$+ \; - \; \times \; /$$

2. Test operations to choose between alternate sets (or"branches") of commands to be executed; the choice depends upon the outcome of an arithmetic test on input data or intermediate results. Any one of the six possible arithmetic relations between two quantities could be tested: less than ($<$), less than or equal to ($\leqq$), greater than ($>$), greater than or equal to ($\geqq$), equal to ($=$), and unequal to ($\neq$).

3. Input and output commands to "read" information into the computer from an external source such as a deck of punched cards and to record the results on an external medium such as a printed sheet of paper or a new deck of punched cards.

Different models of computers have slightly different sets of primitive commands wired in, but these three categories are basic to every modern digital computer.

To perform a computation with a digital computer, a human programmer must first prepare a detailed list of step-by-step commands for the machine. This list of commands, called a program, is "read" into the machine; when the "start button" is pressed, the computer executes the commands in a precisely defined order, operating automatically and without intervention (assuming, of course, that the program contains no catastrophic errors!). The program instructs the machine to read input data, to perform the desired primitive arithmetic operations, to use test operations to perform other operations selectively and repetitively, and finally to print the answers.

The task of preparing such a program for an automatic digital computer is called programming and is the subject of the course described in this Monograph. A computer program must be:

Explicit --	the machine does exactly what the program commands, not necessarily what the programmer meant;"wishful thinking", a common error of beginning programmers, doesn't help.
Detailed --	all the primitive operations must generally be specified, although there are some clever ways of reducing the terrible burden of details in programming.

Precise and unambiguous -- a computer IS only a machine and it blindly executes your program, right or wrong.

We have implied that a computer can perform a different calculation if given a different program; this is certainly true. This is why the machine is called a general purpose automatic digital computer. Some special purpose digital computers have been built in the past with complete programs wired permanently into their circuits; however, they have generally proven to be uneconomical because the simplest change in their "program" requires rewiring.

In principle , a general purpose digital computer can perform (almost) any calculation, given a suitable program to control the execution. There are, of course, practical limitations of computational speed and memory space to keep many problems from being solved on even the largest and fastest computers available today. There are also some theoretical limitations on what a computing machine can compute; a relatively new branch of mathematical logic created in the last few years deals with this question of "computability". But the fact that certain logical problems are in principle unsolvable on a computer has no practical implications for the ordinary numerical computations for which computers are mostly employed today; on the other hand, the practical limitations of speed and size of machine can be very burdensome, indeed.

B. Composing a Program

Most computer programs are quite lengthy and complex; it is not unusual for a program to contain 10,000 or even 100,00 individual primitive commands, each of which must be exactly correct and correctly placed in the sequence of instructions. The mind boggles at the thought of writing and removing the errors from all these commands. Fortunately, the complete program can generally be organized into a series of tasks, each of which has sub-tasks, which in turn have sub-sub-tasks, etc. This hierarchy of tasks is the basic principle of organization for most large programs; without it we could hardly write the complex and lengthy programs which are common today.

But how do we write a program for even the smallest sub-task in the hierarchy? There are two general steps to composing a program:

1. Deciding upon an algorithm.

2. Expressing the algorithm in a programming language for the machine. An algorithm is a complete and unambiguous description

of the steps necessary to perform some (sub) task. Thus, it is a set of instructions, complete and detailed enough for a human -- or even a machine -- to execute. Like a program, an algorithm must be precise and unambiguous.

But how complete is "complete", i.e., how detailed must an algorithm be? Depending upon circumstances, different levels of detail are appropriate in an algorithm. Suppose someone asks for an algorithm to compute the mean of N numbers, and the answer is given, "add them up and divide by N". This answer has probably described the algorithm sufficiently completely for the use of a human (but not a machine). The algorithm could be dressed up with some mathematical notation: "given a set of N numbers, X_1, X_2, ... X_N, the mean is given by the formula:

$$\frac{1}{N} \times \sum_{i=1}^{N} X_i \text{ ,"}$$

but no more detail would have been supplied. In either case, the algorithm has assumed that the operation of adding N numbers:

$$\sum_{i=1}^{N} X_i$$

is primitive.

The most detailed algorithms are those which use only the primitive operations built into a computer; we will call these primitive algorithms. The fact is that the operation:

$$\sum_{i=1}^{N} X_i$$

is not primitive in most computers; in the next section we will give a primitive algorithm for this computation. McCracken in Chapter 1 of A Guide to ALGOL Programming* uses the square root operation as if it were primitive; it is not primitive in most computers, and, in fact, McCracken gives a primitive algorithm for square root in Section 4.1 of his text. Raising an expression to an integer constant power -- e.g., W^3 or $(a + 3b)^2$ -- will be considered primitive, since it is just shorthand for a product which could be written explicitly -- " W x W x W " or "(a + 3b) x (a + 3b)". On the other hand, X^N, where N is a variable, is not

* See Bibliography, Part V.

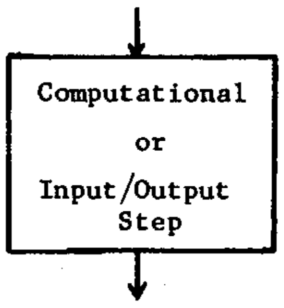
primitive; we would not know how many factors of X to write since the value of N would not generally be known when the algorithm was prepared.

C. Flow Charts - A Language for Expressing Algorithms.

Before we discuss programming languages, we want to clarify the requirements for an algorithm, and particularly for a primitive algorithm; we will therefore take up a language for describing algorithms, known as "flow charts" (also called "block diagrams"). Flow charts are not a programming language, largely because they are two dimensional and pictorial, and there is no convenient way of reading them into the computer. This situation will almost certainly change, since graphical input devices using large cathode ray tubes are now being developed for computers; but for the present, flow charts are limited to expressing algorithms. They are extremely useful for organizing complex programs since the programmer can use any level of detail he wants in his flow charts. For example, he can draw a general flow chart which shows only the outline of the program, with each principal task as a single box; at the other extreme, he can prepare a primitive flow chart, giving the algorithm in complete detail using only the primitive operations of the computer.

A flow chart uses boxes and arrows to show pictorially the "flow", or order of execution, of a program. It has three essential types of components: Computation Boxes, Arrows, and Test Boxes. These will now be described in turn.

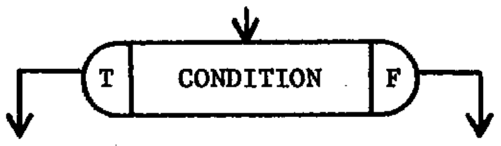
1. A statement, group of statements, or more generally the name or description of a "complete step" in the computation (or an input/output operation) is represented inside a rectangular computation box.



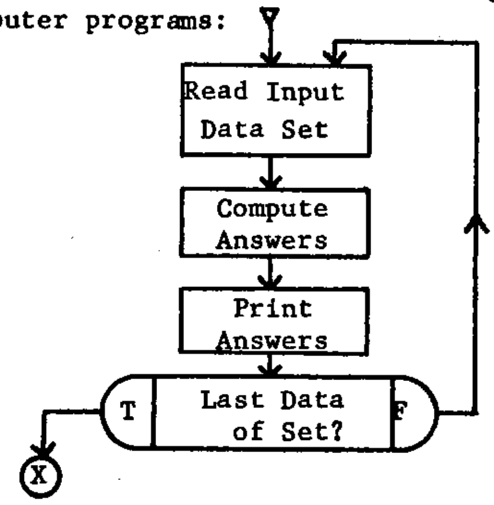
2. A flow chart describes the order of the steps of a computation; arrows are used to indicate the order in which boxes are to be considered and their contents executed. An arrow leads from

each box to its successor, i.e., to the box which is to be executed next. Since a computation box can have only one successor, it will have only one arrow drawn leaving it; however, it can have any number of arrows entering since it can itself be the successor to any number of boxes.

3. The successor can be chosen conditionally by an oval test box. This box contains a condition or test which can be true or false, and has an arrow leaving from each end:



If the condition is true, the successor of the test box is indicated by the arrow from the "T" (true) end; otherwise, the arrow from the "F" (false) end points to the successor. A flow chart should always uniquely specify a successor to each box, or else we have not defined the algorithm sufficiently precisely and explicitly. Here is a very simple example, a flow chart which describes the general algorithm of most computer programs:



Now let us imagine we had a computer which could execute flow charts directly -- i.e., a computer whose primitive machine language was flow charts. The basic execution cycle would be: (1) fetch, from the flow chart stored in the computer memory, the next box to be executed, and (2) execute it; fetch the next box, execute it, fetch the next box, etc. The fetch process would be accomplsihed by following the arrow to the successor box. The computer would start at the "entrance" symbol (Υ) and stop at the "exit" sign (Ⓧ). This alternation of fetching the next command and then executing it, called the "Fetch/Execute" cycle, is fundamental to automatic computers.

We will sometimes find it convenient to indicate arrows implicitly by "labeling" the successor box (we will use a colon to attach the label to the box) and using a "GO TO" box in the form of a circle or oval containing the label.

It is now time to define exactly the form of a primitive flow chart. A computation box of a primitive flow chart contains only basic computational instructions; each of these instructions is executed in two phases:

1. The primitive arithmetic operations +, -, x, and / are performed on variables and numbers, to calculate a new number; and

2. The result is substituted for, or "assigned to", a variable. Note that an algorithm must assign to a variable each intermediate result which is to be saved for later use in the calculation; a computer value can be referred to only if it "has a name".

These two phases are summarized in an assignment statement, which has the following form:

$$\mathcal{V} \leftarrow \mathcal{E}$$

where: $\mathcal{V}$ is any variable name (which may have a subscript),

and $\mathcal{E}$ is any mathematical expression or formula which defines a value to be computed using only primitive operations. We will use $\mathcal{E}$ for such an arithmetic expression.

This is an imperative to compute the value of the expression $\mathcal{E}$, and then assign it to the variable named $\mathcal{V}$. For example, the assignment statement:

$$Y \leftarrow 3A + B^2$$

should be considered as a command to compute the value of the expression $3A + B^2$ using the current values of the variables A and B, and then substitute the result for Y, replacing whatever value Y had previously. The variable Y can have a subscript, as can any of the variables in the expression.

A flow chart contains no statement of facts, only commands. It is important that we recognize the difference between mathematical equations which are passive assertions of fact, and assignment statements which are imperatives.

Mathematics:

$Y = AX^2 + BX + C$	:	This is a statement of mathematical fact. When A, B, C, or X changes, so does Y (or else the assertion becomes false).

Algorithm:

$Y \leftarrow AX^2 + BX + C$	:	This is a command to first compute the value of $AX^2 + BX + C$ and then assign this value to the variable Y. Subsequently changing A, B, or C does <u>not</u> change Y, unless this assignment statement is executed again.

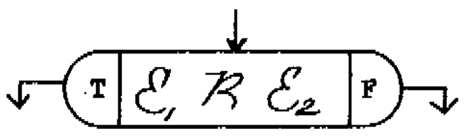
Mathematics:

$I = I + 1$	:	This is nonsense.

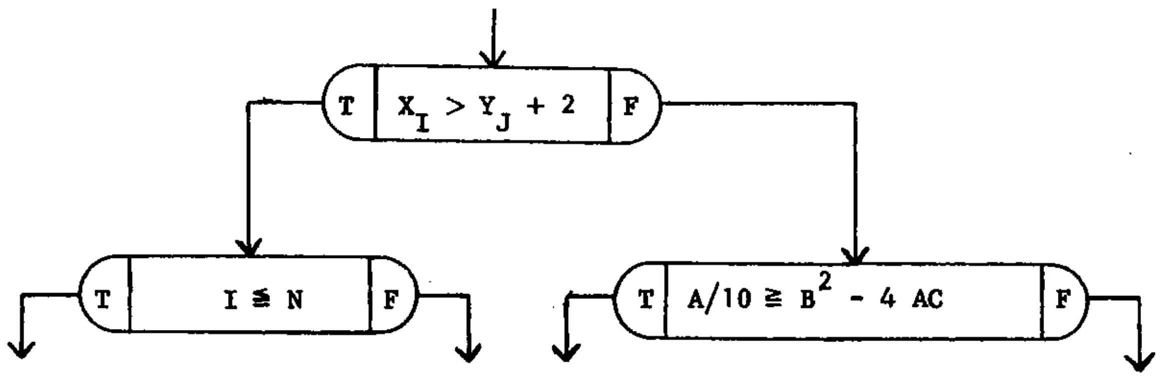
Algorithm:

$I \leftarrow I + 1$	:	This is an operation fundamental to programming since it counts by 1.
$Y \leftarrow \|X\|$	:	This is not primitive, since we did not include absolute value among the primitive operations.

A primitive test box can contain only simple conditions of the form:



where $\mathcal{E}_1$ and $\mathcal{E}_2$ are expressions as defined earlier, and $\mathcal{R}$ is any <u>one</u> of the six possible arithmetic relations: $<$, $\leqq$, $\geqq$, $>$, $=$, or $\neq$. For example, the following test boxes are primitive:

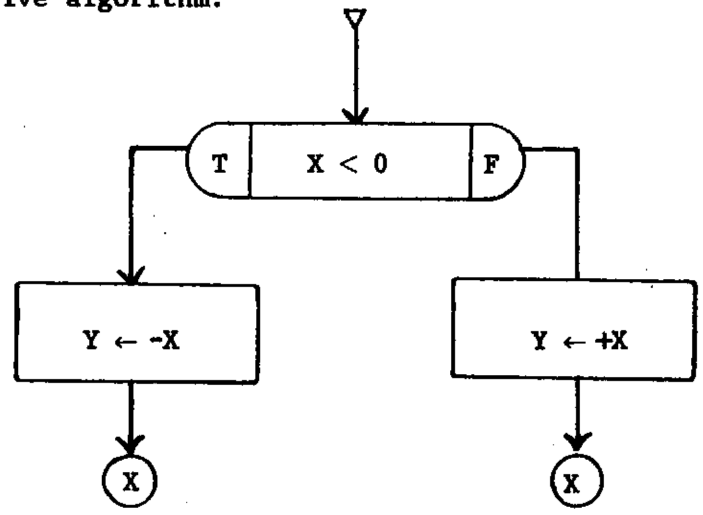


D. Examples of Algorithms as Flow Charts.

The first two examples we will give, Absolute Value and Truncation, are built into many computing machines as primitve operations, and could therefore be considered to be primitive in flow charts.

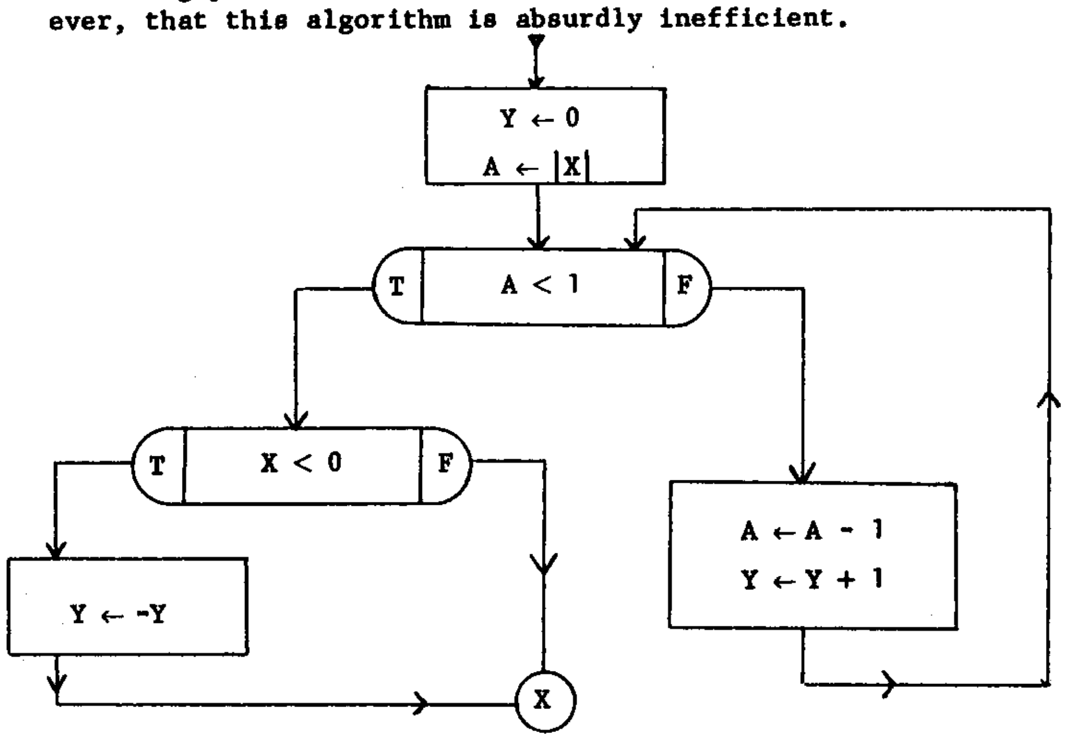
EXAMPLE 1: Absolute Value.

If $\boxed{Y \leftarrow |X|}$ is not primitive, then it can be found from the primitive algorithm:



EXAMPLE 2: Truncation.

The Truncation operator is symbolized by a down arrow ("$\downarrow$"); it removes the fractional part of a number, leaving only the integer part. For example, $\downarrow(3.76) = 3$, and $\downarrow(-3.76) = -3$. To make a primitive flow chart for truncation, we must subtract 1 repeatedly until the result is less than 1; therefore, our algorithm needs a loop, which is traversed repeatedly until a condition is met. The following primitive flow chart will compute: $Y \leftarrow \downarrow X$; note, however, that this algorithm is absurdly inefficient.



EXAMPLE 3: Summing N numbers.

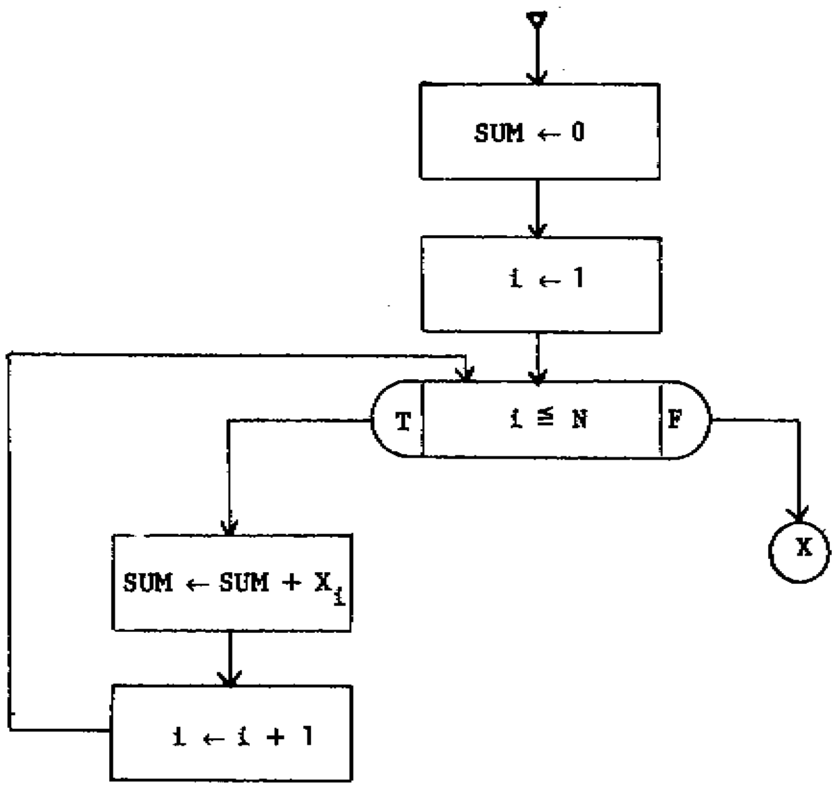
Suppose we are given a set of N numbers, in the variables X_1, X_2, ..., X_N, and we need to compute their sum:

$$Y = \sum_{i=1}^{N} X_i = X_1 + X_2 + \ldots + X_N$$

If N is a small constant, we can write a single assignment statement; for example, if N = 3, the complete algorithm is:

$$\boxed{Y \leftarrow X_1 + X_2 + X_3}$$

But suppose N is large so that we don't want to write out all the terms, or, as is common in programming, the value of N is variable and may be different each time the algorithm is executed. Unfortunately, we can't use the three dots of mathematics in an algorithm or a program since this is not explicit enough for a machine. Instead, we must use a loop. The following loop will work:



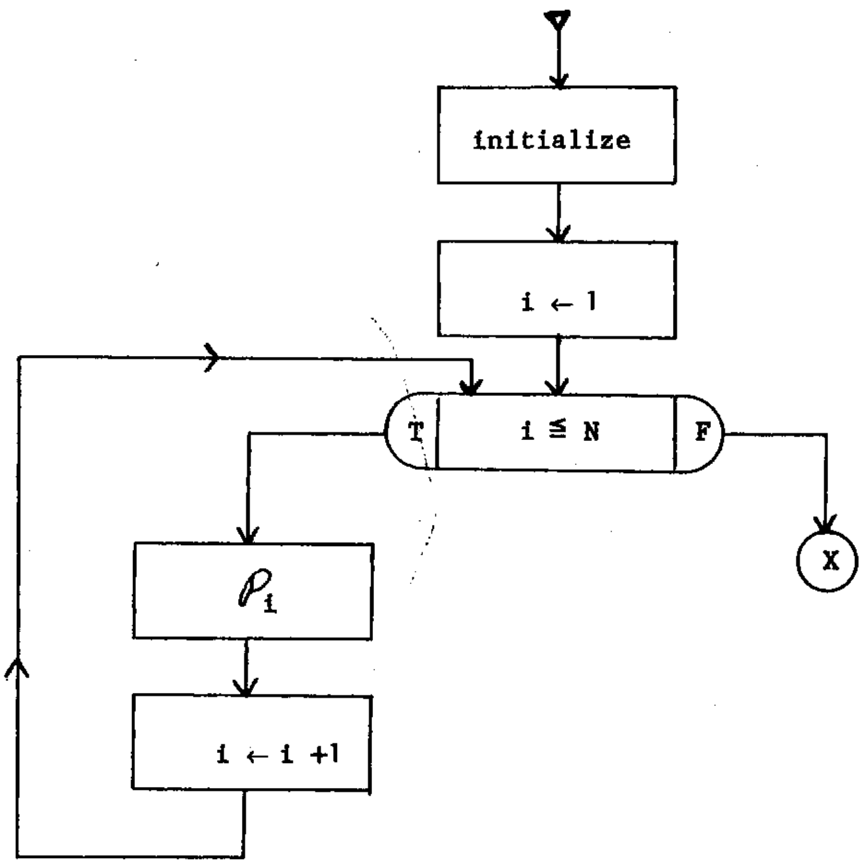
This algorithm can be explained in the following way: it executes the process: "add X_i to the sum-so-far" (i.e., "SUM ← SUM + X_i") N times, with i taking each of the values 1, 2, 3, etc., up to (and including) N. The result is to compute the sum of the N numbers and leave the result in the variable SUM.

Without explaining its exact meaning at this state, we can point out the plausible ALGOL notation which is equivalent to this flow chart:

```
          SUM ← 0 ;
          for i ← 1 step 1 until N do SUM ← SUM + X[i];
```

This can be read as follows: "assign 0 to SUM; then for values of i starting at 1 and increasing in steps of 1, until N is reached, do the assignment: SUM ← SUM + X_i".

The flow chart for summing N numbers is a particular example of a very general class of algorithms containing a loop. We can abstract the general features of all of these flow charts by assuming that there is some "basic process" depending upon i and symbolized by $\mathcal{P}_i$, to be executed for i = 1, 2, ..., N:



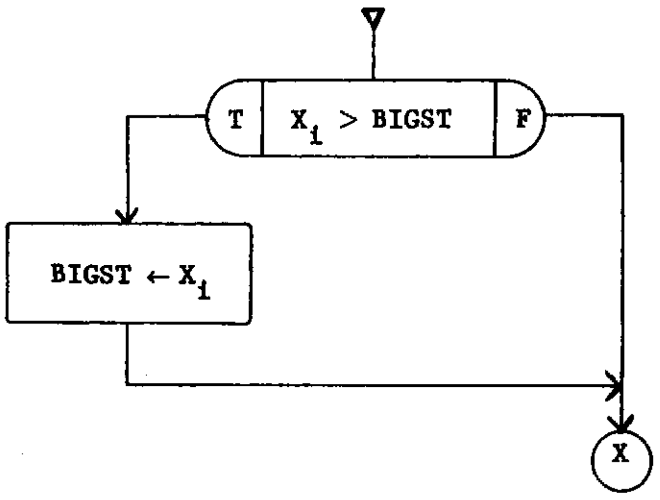
The box containing " $\mathcal{P}_i$" can represent any complete sub-flow chart. Notice carefully the general features of this generic flow chart:

(1) First an "initialization box" sets things up.
(2) A test is made for completion; if N = 0, one usually doesn't want $\mathcal{P}_i$ executed even once, so it is better to test for completion before going to step (3).
(3) Execute the basic process $\mathcal{P}_i$.
(4) Step to next i value, and go back to (2).

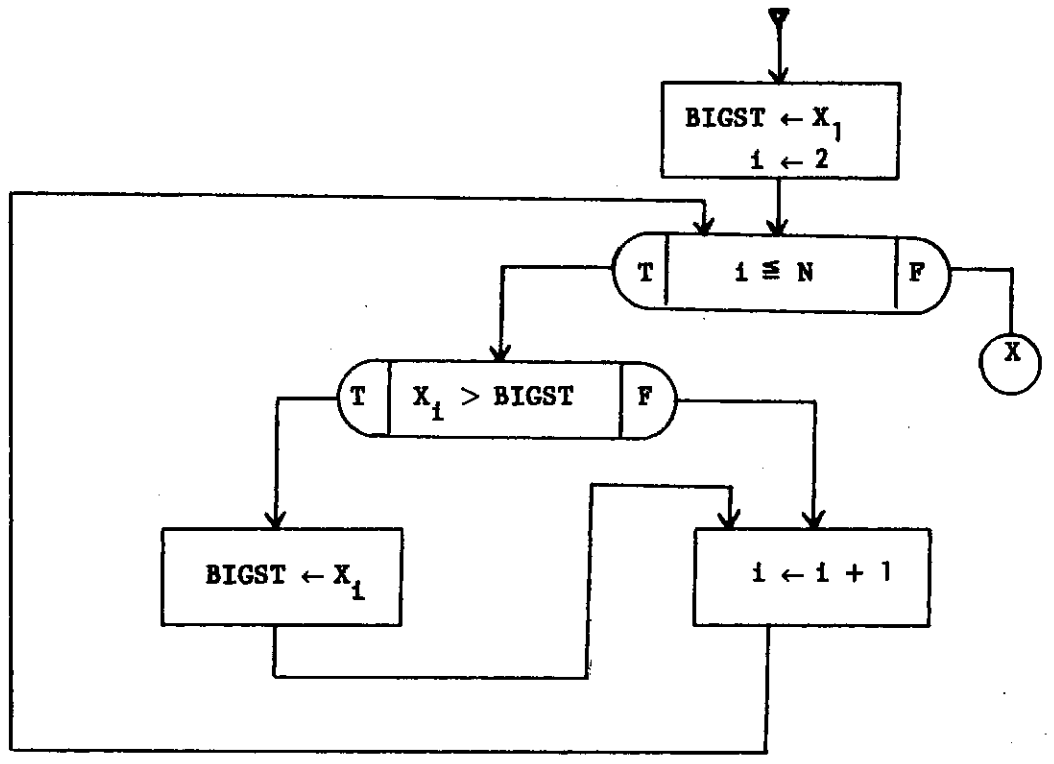
This general loop flow chart can help in developing particular algorithms; one answers the question, "what basic process, if executed for i = 1, 2,...,N, will accomplish the desired task?" This is essentially an algorithm for making up algorithms, and is a very useful approach, as we will now illustrate.

EXAMPLE 4: Finding the Largest of N Numbers.

Suppose we wish to find the largest (the "maximum") of a set of N numbers: X_1, X_2,...,X_N. Since N is a variable, we need to apply the general loop flow chart. The following basic process will find the maximum if executed for i = 1, 2, 3,..., N: "If X_i is larger than the biggest-found-so-far, then use X_i as the biggest-found-so-far". If the variable BIGST contains the "biggest-found-so-far", this basic process is defined by the following flow chart:



Note that the entrance ("Y") and exit ("(X)") symbols stand for the "local" entrance and exit of this sub-flow chart, not the entrance and exit of the bigger flow chart of which it is a single box. Thus, we will generally assume that every flow chart (call it "F_1") only represents a single box of a larger, less detailed, flow chart (call it "F_0"); then the entrance symbol in F_1 stands for arrows in F_0 from predecessors of the F_1 box, while the exit symbol in F_1 stands for arrows to the successor in F_0. Substituting the sub-flow chart for BIGST into the general loop gives a complete algorithm for finding the largest value of the N numbers.



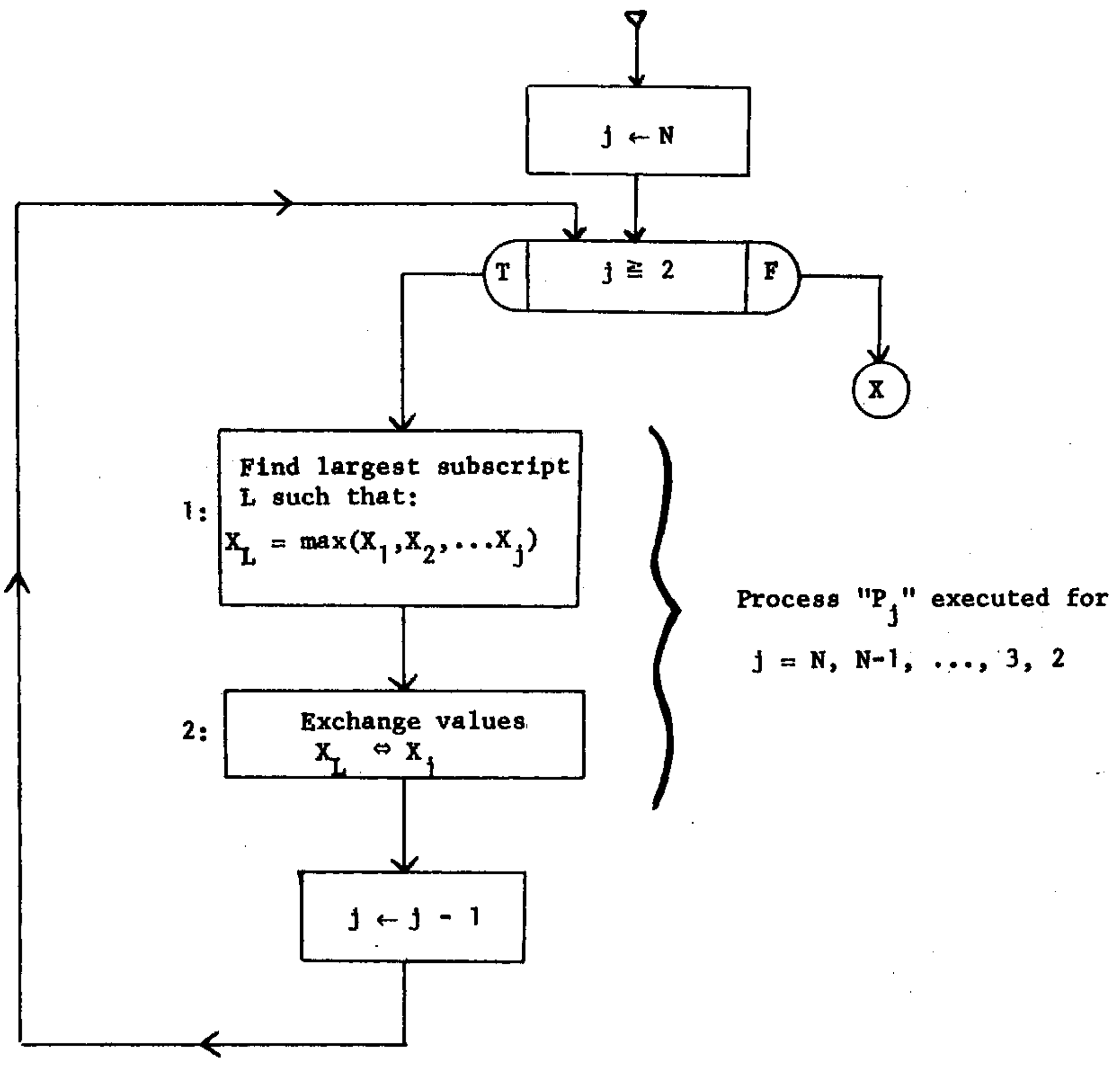
We have initialized BIGST to X_1, and started the loop with $i = 2$ instead of $i = 1$.

EXAMPLE 5. Sorting N Numbers into Ascending Order.

A very simple (but inefficient)method of sorting N numbers $X_1, X_2, \ldots, X_N$ into order is the following: find the largest of the N numbers and interchange its value with X_N, so the largest is at the end of the list where it belongs; then find the largest of $X_1, \ldots, X_{N-1}$, and exchange its value with X_{N-1}; this is continued until all are sorted. It is convenient to consider the basic process to be executed for $j = N, N-1, N-2, \ldots, 3, 2$; the process is:

> First, find the (largest) subscript L such that X_L is the largest among $X_1, \ldots, X_j$ (the unsorted values); then interchange the value of X_L with the value of X_j.

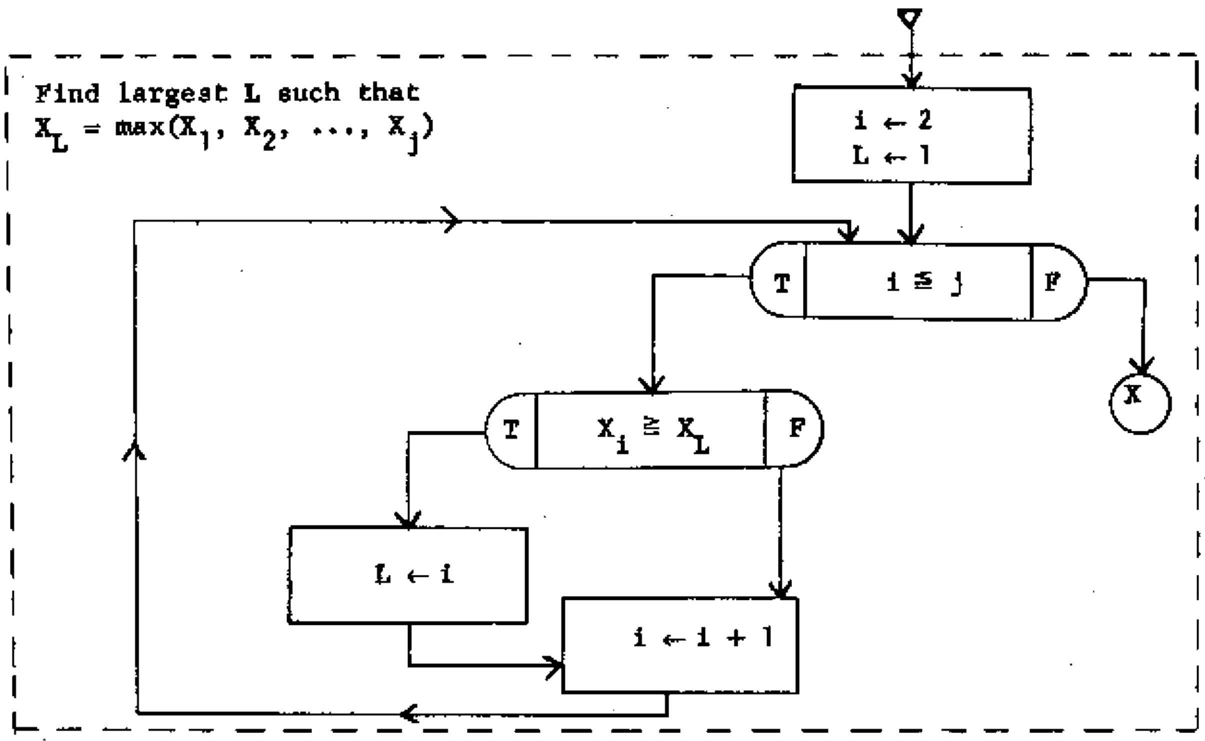
Thus, the sorting algorithm is given by the following flow chart:



$j \leftarrow N$
T
$j \geq 2$
F
X
1:
Find largest subscript L such that:
$X_L = \max(X_1, X_2, \ldots X_j)$
2:
Exchange values $X_L \Leftrightarrow X_j$
$j \leftarrow j - 1$
Process "P_j" executed for
$j = N, N-1, \ldots, 3, 2$

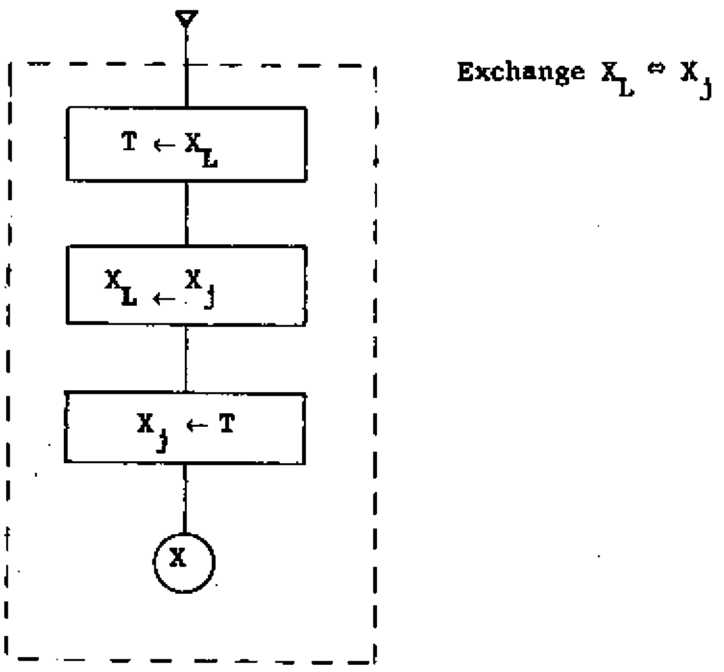
This is not a primitive flow chart since the boxes labeled 1 and 2 are not primitive. We can give primitive flow charts for each of them, however. Box 1 is essentially the algorithm of EXAMPLE 4, except that we are now finding not the maximum value, but the subscript of the maximum value; X_L takes the place of the variable BIGST.

1:



Finally, the algorithm of box 2 requires no loop; however, an extra temporary variable T is needed if the two values are to be <u>interchanged</u>.

2:



E. The Dual Nature of ALGOL.

We have discussed the use of flow charts for expressing algorithms. There is another language suitable for algorithms: ALGOL. Unlike flow charts, ALGOL is designed to be not only an algorithmic language but also a programming language; that is, an algorithm written in ALGOL can be executed on a computer as a program.

This dual purpose of ALGOL is the fundamental reason for the importance and usefulness of this language. Unfortunately, the needs of an algorithmic language and a programming language conflict somewhat, and therefore ALGOL is not a perfect language for either application. ALGOL is generally slightly more verbose and less concise than one sometimes desires in a programming language, in order that ALGOL programs, or algorithms, can be easily and clearly read by people. However, considered in its dual role, ALGOL is the best algebraic computer language available today.

One of the most important uses of ALGOL is for the international publication of algorithms. There are generally many different algorithms known for most computing tasks, all differing in speed, computer memory requirements, and accuracy. Many of these algorithms for important computing tasks are being published along with comments on experience with them, in the ALGOL language; the publication of algorithms is one of the important results of the development of ALGOL in 1960. From these published algorithms one can build a library of tested, certified programs for a great variety of tasks, which can be called and used by one or two ALGOL statements.

Even if one does not have an ALGOL compiler available and uses a different language for programming the computer, it is still important to know ALGOL in order to be able to read published algorithms (or even to publish them oneself). At the very least, the published algorithms are a valuable source of ideas and techniques.

F. The Syntactic Structure of ALGOL.

If we analyze the structure of a page of English prose, we find letters formed into words which in turn are formed into sentences and into paragraphs, etc. An ALGOL program is built up in an analogous manner from the basic characters, which are put together to form numbers and variables, to form expressions, and finally statements and declarations. Let us briefly describe each of these levels of grammatical structure of an ALGOL program.

1. Numbers. We have been using numbers, that is, constants, without comment in our flow chart. There are specific rules

for forming correct numbers in ALGOL; see chapter 2 of McCracken's text. The following are examples of correct numbers in ALGOL:

$137.05,\ 1,\ 6.02_{10}23$

2. Variables. Both simple and subscripted variables can be used in an ALGOL program, with the same meaning that they had in flow charts. I, BIGST, and X1 are examples of ALGOL simple variables. Since ALGOL programs must be punched into cards, a subscript cannot be lowered below the line but instead must be inside square brackets. For example,

   X[1], X[I], and A[I, J+2, K]

   are ALGOL subscripted variables.

   The name of a variable in ALGOL is called an identifier; in fact, all names in ALGOL programs are identifiers. An ALGOL identifier must have the form of a letter, possibly followed by one or more letters or digits. The identifier "X1" could represent a simple variable; the "1" is part of the identifier (that is, the name) of the simple variable; however, "X[1]" is a subscripted variable, "X-sub-one". When we write an ALGOL program,the identifiers can be chosen as we please; however, the same identifier may not be used for both a simple variable and a subscripted variable in the same program (or, more precisely, in the same block).

3. Expressions. We define an expression as: a rule for computing a value. There are different kinds of expressions, with different kinds of values. The most obvious kind of value is arithmetic, and ALGOL contains arithmetic expressions. The complete rules for forming ALGOL arithmetic expressions are discussed in Chapter 2 of McCracken's text.

   In the test boxes of primitive flow charts we had conditions, such as "$X > Y$". Such a condition is really an expression which defines a "truth" value: true or false. An expression which has a true or false value is called a Boolean expression. A condition is not the only form of Boolean expression; in Chapter 3 of his text, McCracken shows that much more complex Boolean expressions are possible.

4. Statements. A statement in ALGOL is a complete command, and generally corresponds to a computation box in a flow chart. For example, " I := I + 1" is an ALGOL assignment statement.

Assignment statements in ALGOL have the same form as those we have been using in primitive flow charts. The assignment operator in ALGOL-60 is " := "; however, in this monograph we will use the symbols " := " and " ← " interchangeably.

Another example of an ALGOL statement is the for statement:

$$\underline{\text{for}}\ I \leftarrow 1\ \ \underline{\text{step}}\ \ 1\ \ \underline{\text{until}}\ \ N\ \ \underline{\text{do}}\ \ SUM \leftarrow SUM + X[I]\ .$$

A for statement provides a short and concise way of writing a loop in a program; however, the complete rules for for statements are complex. This is the subject of Chapter 5 of McCracken's text.

5. Declarations. A flow chart representing an algorithm contains commands which are to be executed in a particular order. This is generally all that is necessary to express an algorithm. However, ALGOL is not only an algorithmic language but also a programming language; that is, an ALGOL program can be executed on the computer. As a result, the programmer has to put some information into his ALGOL program which is not strictly part of the algorithm. This information, contained in ALGOL declarations, is necessary so the program can be executed more efficiently, and so choices can be made about the allocation of the machine's memory to the data of the program, about the accuracy of the arithmetic, etc. Although ALGOL statements are commands, ALGOL declarations are simply assertions of fact; that is, they are passive; it is important to understand this distinction between statements and declarations.

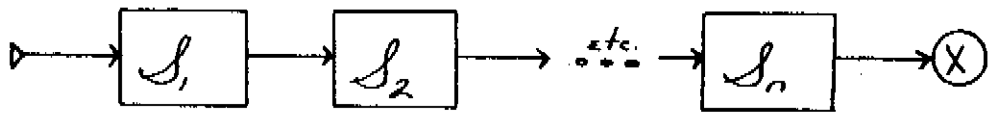
Let us consider now the forms of ALGOL statements and the relation of ALGOL statements to flow charts. That is, an entire ALGOL program could be thought of as one long string of characters, formed into expressions and then into statements. A program is basically a series of statements written one after another and separated by semi-colons. Thus, if we represent a general ALGOL statement by "$\mathcal{S}$", then every ALGOL program has the form*:

$$\mathcal{S}_1\ ;\ \mathcal{S}_2\ ;\ \circ\circ\circ\ ;\ \mathcal{S}_n$$

To execute the ALGOL program, the computer executes each statement in turn in left-to-right order, starting with the leftmost statement. If each statement $\mathcal{S}_i$ corresponds to the contents of one computation box $\mathcal{S}_i$ then this statement sequence is identical in meaning to the

* The three open circles are an ellipsis symbol.

following flow chart:



Although an ALGOL program is logically a one-dimensional string of characters, most programs are too long to fit on one line. Therefor the program can be punched into any number of cards; the first column of each card follows logically immediately after the last column of the preceding card, so that the program is still effectively a long-dimensional string of characters. Several short statements may be punched on one card, or one statement may be punched on each card, or one long statement may be spread over many cards, whichever gives the most readable program. The "machine" (i.e., the ALGOL translator) ignores all the blank spaces and treats the program as one long continuous string of characters. Therefore when we speak of "left-to-right order", it may actually be "top-to-bottom order" as punched on the cards.

Since blank spaces are irrelevant to the meaning of an ALGOL program, many extra blanks can be inserted to space the statements on the cards in such a way that the program is very readable. Consistent use of this freedom to make an ALGOL program readable and transparent is very important if ALGOL is to perform its function of communicating algorithms to other people.

G. Exercises on Algorithms and Flowcharts

<u>Exercise 1</u>:

For each of the following computational tasks, draw a primitive flow chart representing an algorithm to perform the task.

(a) Assume you are given two numbers X and Y. Compute P by the rule:

$P \leftarrow 1$ if X and Y are both less than 6, or if X and Y are both greater than 6;

$P \leftarrow 0$ otherwise.

(b) You are given a set of N numbers $X_1, \ldots, X_N$; assume they have been sorted into ascending order, so that:

$X_1 \leq X_2 \ldots \leq X_N$. Compute the median M by the rule:

If N is <u>odd</u>, then

M ← (the "middle" X value, such that half of X's are ≦ M and half are ≧ M);

If N is <u>even</u>, then

M ← (the average of the two middle values).

Suggestion: in this problem, use the truncation operator (" ↓ "), assuming it to be primitive (see Example 2).

(c) Given a set of N numbers $X_1,\ldots,X_N$, count how many of them are positive (>0). Call your result P.

(d) Assume you are given values of N and $X_1, X_2,\ldots,X_N$. Find the smallest subscript L such that X_L has the largest <u>absolute</u> value of any of the X_i's; i.e.,

$$X_L = \text{Max} (|X_1|, |X_2|,\ldots, |X_N|).$$

In this flow chart, assume that absolute value is <u>not</u> a primitive operator.

(e) Suppose you are given a set of N non-negative numbers $X_1, X_2,\ldots,X_N$, and you wish to compute a new set of numbers $f_0, f_1, f_2,\ldots,f_{99}$ which form a "<u>frequency count</u>" of the X's. That is, f_i will count how many X's are in the range $i \leqq f_i < i+1$, for integers $i = 0, 1,\ldots,99$. Ignore X's which are >100. You may use " ↓ " as a primitive operator.

(f) Given any real value X and any <u>integer</u> value N, perform the operation:

$$P \leftarrow X^N$$

by repeated multiplications (or divisions). If $N \leqq 0$ and $X = 0$, then execute the box:

$$\boxed{\text{PRINT "Error"}}$$

and set $P \leftarrow 10^{50}$ to represent "infinity".

Comments on this problem:

To extend this algorithm to compute X^Y where X and Y are any (real) values, you would need the functions e^X and $\log_e(X)$; these functions are not primitive, of course, but can be computed to any desired accuracy with primitive algorithms. Then combining all these flow charts, you could draw a single primitive flow chart for X^Y. The algorithm for this flow chart has been programmed in the computer's primitive machine language in the form of a "sub-program" or "subroutine". In the ALGOL language, the operation X^Y (written "X↑Y") is considered primitive; an ALGOL program which uses this operation simply executes the entire X^Y sub-program. This is an example of a general principle in programming -- once an algorithm has been written as a sub-program, this program can be treated as a new "primitive" process, effectively enlarging the instruction repertoire of the computer. This process of defining new operations can be carried to any number of levels; for example, sub-programs can be defined for e^X and $\log_e(X)$, which can be used in turn as "primitive processes" in defining the X^Y algorithm, which can be used as a "primitive process" in defining other functions. At the very bottom of the hierarchy, of course, there must be "truly primitive" algorithms which use only the operations built into the computer.

(g) Suppose one needs to compute cos x, accurate to 10 decimal places, for any angle x.

(a) If x is a very small angle, $|x| < 10^{-3}$, then the simple formula:

$$\cos x = 1 - \frac{x^2}{2}$$

is accurate to better than 12 decimals.

(b) Given cos x for any angle x, cos 2x is given by:

$$\cos 2x = 2(\cos x)^2 - 1$$

Draw the primitive flow chart to compute $Y \leftarrow \cos x$ for any angle x, using (a) and (b). Simply halve the angle repeatedly until it is small enough to apply (a). Then apply (b) often enough to get the cosine of the original angle.

<u>Exercise 2</u>:

Draw a flow chart for an algorithm which: (1) determines if a given sequence of 0's and 1's contains at least one occurrence of the sequence 101 and (2) sets a Boolean variable TEST to <u>true</u> if (and only if) 101 does appear. The end of the sequence will be marked by a 2 and no other 2's will appear.

For example, the sequence

$1100\underline{101}1\underline{101}2$

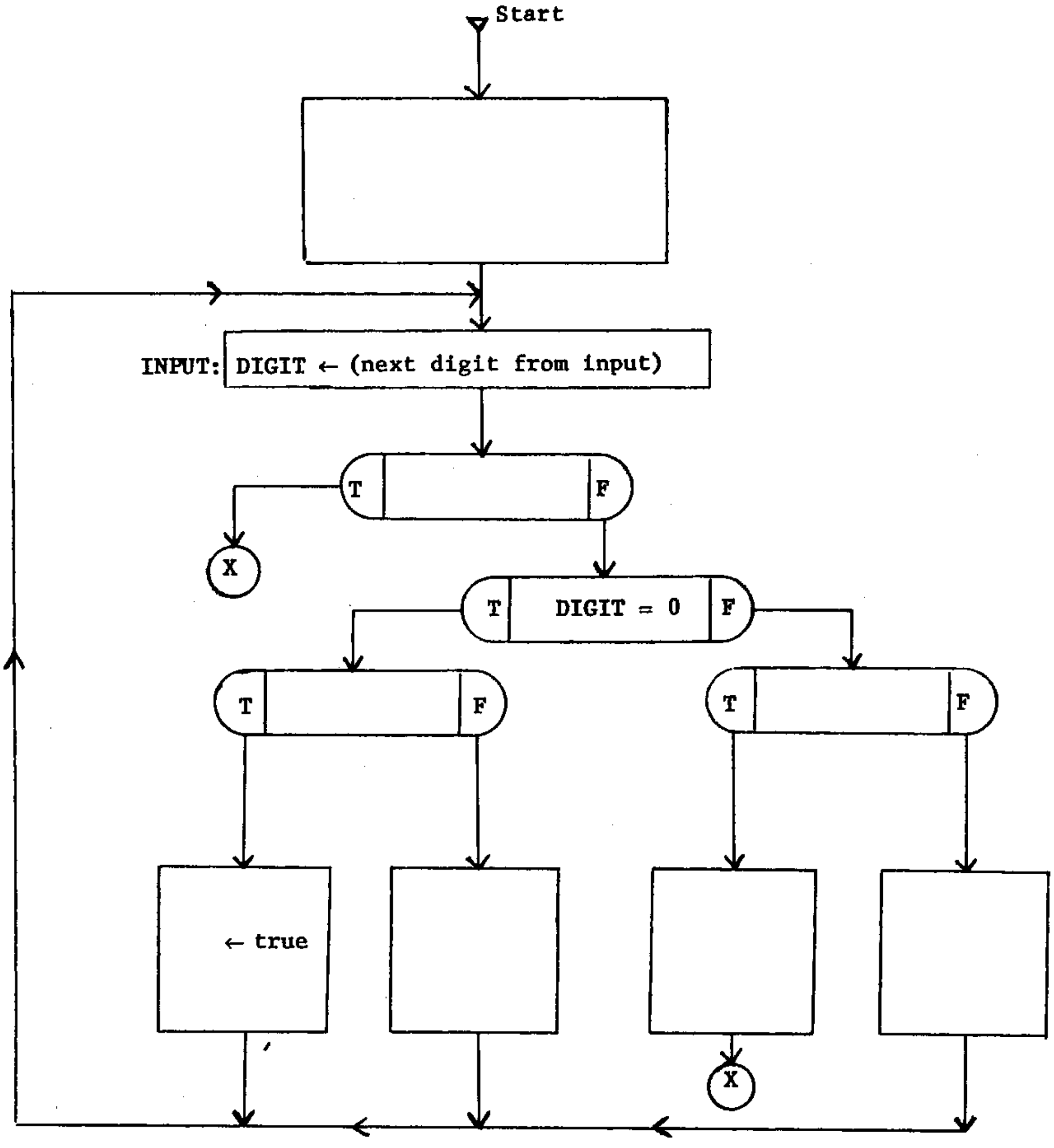
does contain the sequence 101 (twice) while the sequence

1100100112

does not.

On the next page is a flow chart which has some empty boxes. You are to fill in these empty boxes so that the flow chart gives an algorithm to solve the problem. You may use no other variables except DIGIT which holds the next input character, the answer TEST and two Boolean variables ONE and ONEZERO. Test boxes may contain only Boolean expressions, and rectangular boxes may contain only assignment statements. Use all boxes and do not add any boxes. Only in the box marked INPUT may a character of the input string be obtained.

Exercise 2: Flow Charts (cont'd)

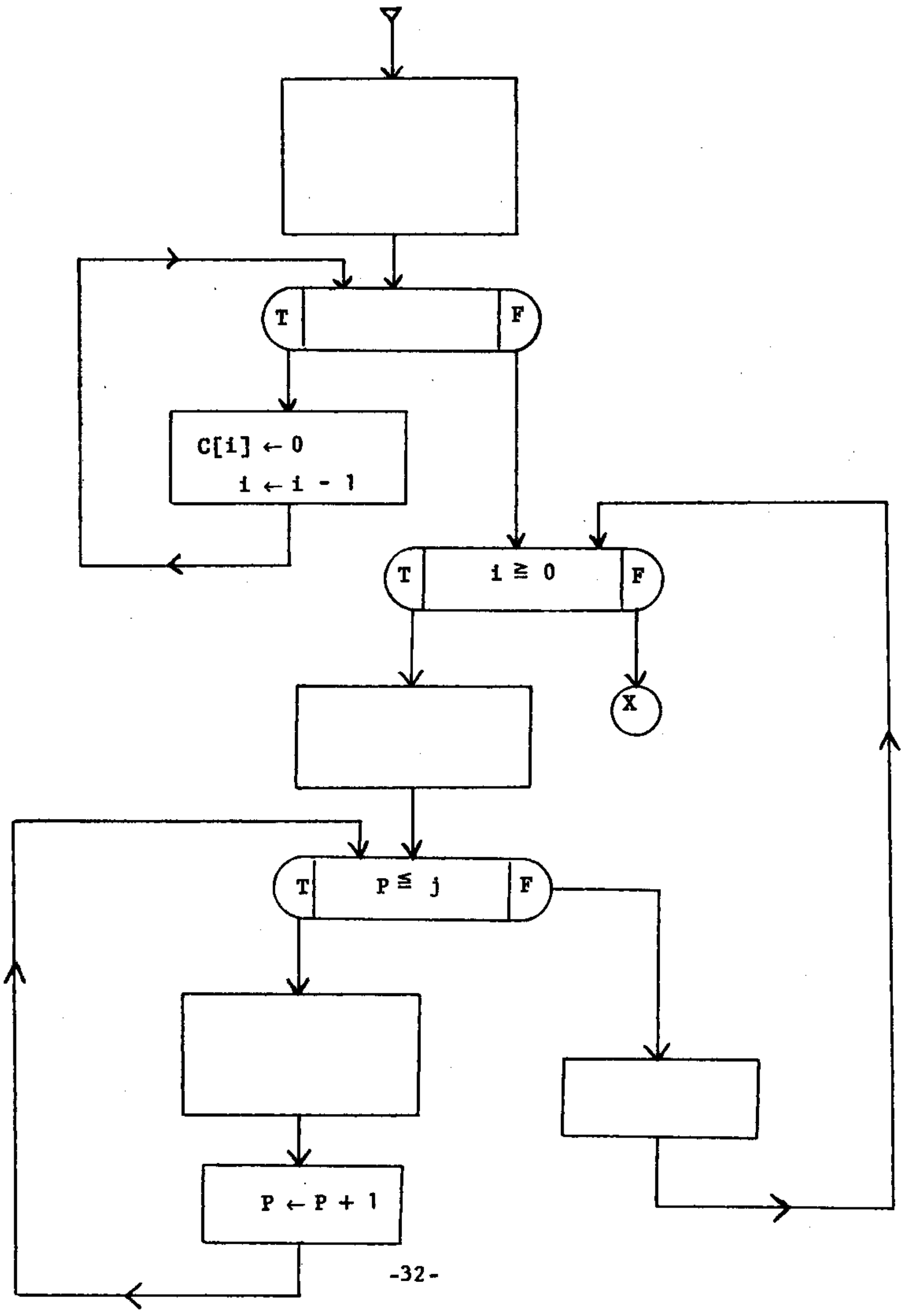


Exercise 3:

$$\text{Let: } A_m(X) = \sum_{i=0}^{m} X^i a_i \text{ and } B_n(X) = \sum_{i=0}^{n} X^i b_i$$

be polynomials in X of degree m and n, respectively. Complete the flow chart on the next page for an algorithm to compute the coefficients $C_0,\ldots,C_{m+n}$ of the polynomial product of $A_m(X)$ and $B_n(X)$. Your algorithm should be such that if it were transformed into a computer program, the same memory cell could be used for C_j and b_j, for each $j = 1,2,\ldots,n$; however, the a_i's and b_j's must occupy distinct cells.

Fill in the missing statements but don't add any boxes or lines. Note: no additional subscripted variables are necessary in this problem.


T
F
C[i] ← 0
i ← i - 1
T
i ≧ 0
F
X
T
P ≦ j
F
P ← P + 1

__Exercise 4__:

Consider a sequence of n numbers: $v_1, v_2, \ldots, v_n$. We are interested in sets of consecutive v's which have the value 0, with no non-zero values intervening; these sets we will call "runs of zero" in the v sequence. The number of zero values in each run of zeros is the "length" of the run. A run of zeros is bounded on each end by either a non-zero v or an end of the complete v sequence.

Example: The sequence of v's: $\underbrace{0},1,\underbrace{0,0,0},1,1,\underbrace{0,0,0},1$ contains three runs of zeros: two runs of length 3, and one of length 1. If we also say that each non-zero value is a "run of zeros of length 0", then there are four runs of length 0.

Problem: compute the frequency of occurrences of runs of each possible length; i.e., compute f_0, f_1, $f_2, \ldots$, where f_i is the number of runs of zeros of length i in the given sequence of v's. Assume that the f_i's have already been set to zero before reaching this flow chart.

Complete the primitive flow chart on the next page for an algorithm to compute the f_i's. No additional boxes or arrows are needed. Some computation boxes may contain more than one assignment statement, but every box will contain at least one statement. Note: this problem is not trivial, but it does have several solutions.

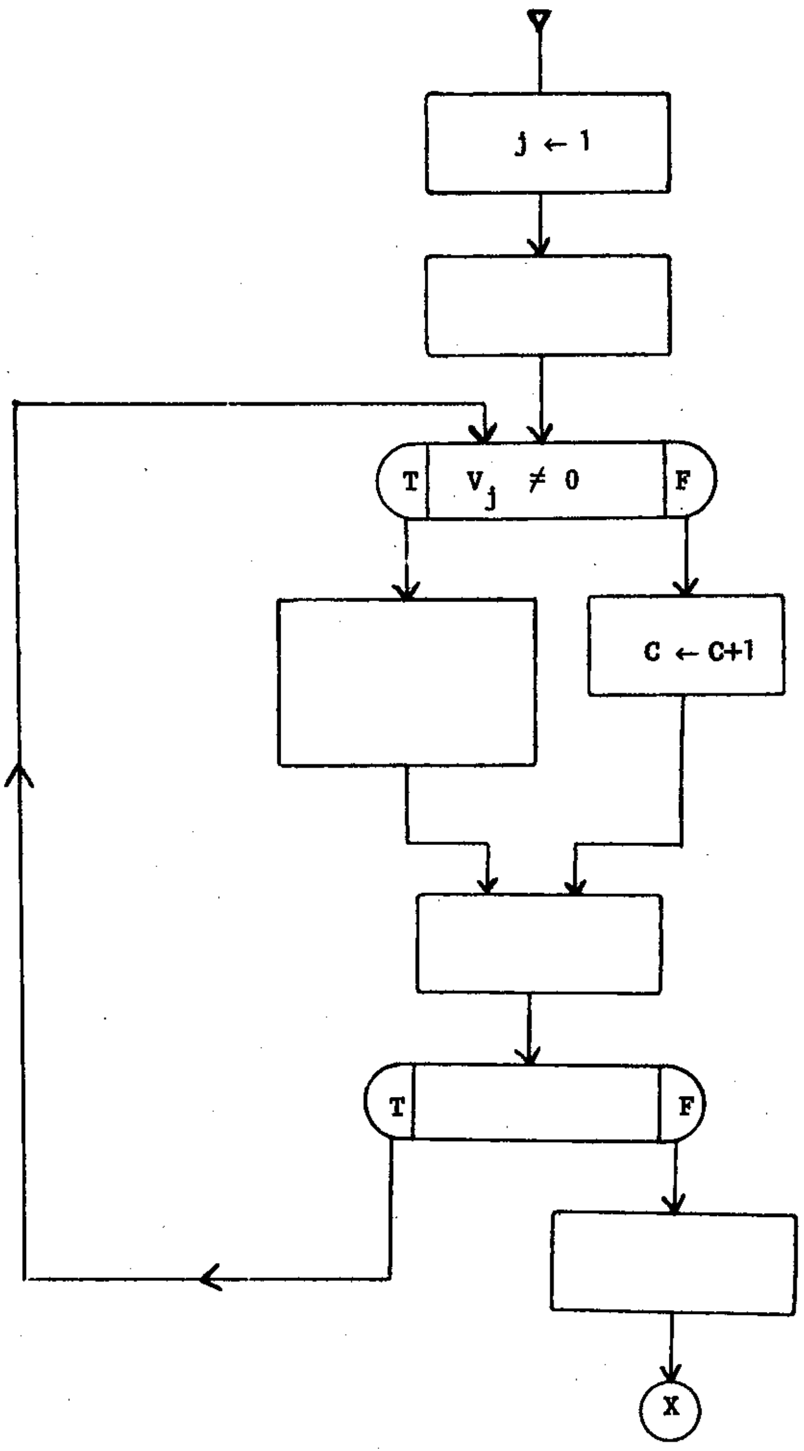
__Exercise 4__:

Consider a sequence of n numbers: $v_1, v_2, \ldots, v_n$. We are interested in sets of consecutive v's which have the value 0, with no non-zero values intervening; these sets we will call "runs of zero" in the v sequence. The number of zero values in each run of zeros is the "length" of the run. A run of zeros is bounded on each end by either a non-zero v or an end of the complete v sequence.

Example: The sequence of v's: $\underline{0},1,\underline{0,0,0},1,1,\underline{0,0,0},1$ contains three runs of zeros: two runs of length 3, and one of length 1. If we also say that each non-zero value is a "run of zeros of length 0", then there are four runs of length 0.

Problem: compute the frequency of occurrences of runs of each possible length; i.e., compute f_0, f_1, $f_2, \ldots,$ where f_i is the number of runs of zeros of length i in the given sequence of v's. Assume that the f_i's have already been set to zero before reaching this flow chart.

Complete the primitive flow chart on the next page for an algorithm to compute the f_i's. No additional boxes or arrows are needed. Some computation boxes may contain more than one assignment statement, but every box will contain at least one statement. Note: this problem is not trivial, but it does have several solutions.


j ← 1
T
$V_j \neq 0$
F
C ← C+1
T
F
X

2. DATA STRUCTURES: DATA ORGANIZATION AND REPRESENTATION.

A. Arrays

There are two principal intellectual tasks in planning a complex computer program: the organization of the program itself, and the choice of the best organization and representation for the data. By data we mean all the numbers (in particular, variables) stored in the machine's memory which are not themselves part of the program (i.e., are not executed) but are operated upon by the program. Later we will discuss program structures, and will point out that the best way to organize a complex program is (almost) always as a hierarchy of subroutines. In this chapter we will consider ways of structuring and accessing the data in the computer.

ALGOL is designed for describing and programming complex calculations on relatively simple data structures: arrays of subscripted variables. The idea of a subscripted variable is, of course, lifted directly from mathematical notation. Thus, in mathematics we often denote an entire set of related variables $V_1, V_2, \ldots, V_n$ by a common name (e.g., "V") and use a subscript (e.g., "2", or "i") to pick out a particular variable from the set. Such a set of n variables, distinguished by a single subscript, can be referred to as a "vector".

We may attach two subscripts to a set of variables: for example, a_{ij}, where the two independent subscripts may take on values: $i = 1, 2, \ldots m$, and $j = 1, 2, \ldots n$. Such a set of $m \times n$ variables can be represented pictorially as a two-dimensional array:

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \qquad \text{(A1)}$$

By convention, the first subscript is the row index and the second subscript is the column index of an element a_{ij}; hence this matrix has m rows and n columns.

Finally, mathematics considers sets of quantities with any number of subscripts. The simplest structure of all is, of course, a "simple variable" or "scalar" which has no subscripts.

The subscripted variable of mathematics has been adopted into programming as an array of variables, called simply an array. Following the idea of the two dimensional picture of a matrix, we can refer to each subscript of an array as a dimension, so that we can have arrays of one, two , three or generally any number of dimensions. Paralleling the mathematical data structures: scalar, vector, and matrix, therefore, we have the program data structures: simple variable, one-dimensional array, and two-dimensional array, respectively. We could furthermore call a simple variable "an array with no dimensions" (i.e., no subscripts).

In ALGOL, the particular subscripts which distinguish a subscripted variable are written after the array name and on the same line, with the subscripts surrounded by square brackets; thus, the form* of an ALGOL subscripted variable is:

<array name> [<subscript>, <subscript>, ... , <subscript>] (A2)

A <subscript> can be any arithmetic expression , such as "2", or "I", or "I + 2", or "A[A[I + 1]-3]/2". A subscripted variable from an array with n dimensions must always have n <subscript>s ; this is true for all values of n including n = 0, so that the same variable cannot be used both as a simple variable and as a subscripted variable.

The number of dimensions in an array must be specified in an array declaration. The declaration furthermore indicates for each dimension the upper and lower bounds for the corresponding subscript, in the form:

array <name> [<lower bound> : <upper bound>, ... , <lower bound> : <upper bound>] . (A3)

For example, the declaration: "array Q[1:m, 0:n+1]" is short for the assertion: "Q is a two-dimensional array whose first subscript runs from 1 to m and whose second subscript runs from 0 to n + 1". We will use ALGOL declarations in the text of this chapter as an abbreviated notation for such an assertion about an array name.

We can distinguish three different aspects of the use of data structures in programming: abstract structure, representation, and allocation. In this section we have discussed the abstract structure of array data structures; by "abstract" we mean: independent of the

---

* Note: We are using the meta-brackets < and > of BNF here. For an explanation of their meaning, see Section 4 of Part II. The three open circles "..." are an ellipsis symbol.

manner in which the arrays are represented in the machine's memory and of the way that the memory space is allocated to the structure. Section C and D will investigate the representation of arrays in memory; a complete discussion of allocation of space would require an understanding of ALGOL block structure.

In Section E we will return to a consideration of abstract data structures, and introduce some classes of structures which are more general than arrays. Representation and allocation for these structures will be discussed also.

The next section contains some simple examples of important ways of using arrays in programming.

B. Programming With Arrays

Suppose we have a program which must evaluate the factorial function n! many times in the course of execution. An obvious approach is to write a procedure (subroutine) to evaluate n! , in the following manner:

```
real procedure FACTORIAL  (N) ;                              (B1)
          begin  real F; integer I;
               F ← 1; for I ← 1 step 1 until N do F ← F * I;
               FACTORIAL ← F
          end ;
```

If this procedure is called many times, however, a great deal of redundant calculation is performed. A more efficient program results from computing a table of n! once at the beginning of the program and storing the values in an array: array FACT [0 : m], in such a way that FACT[i] = i!. This array of values could be generated by the algorithm:

```
          FACT [0] ← 1 ;                                     (B2)
          for I ← 1 step 1 until m do
                    FACT [I] ← I * FACT [I - 1];
```

Once this table is generated, values of n! can be "computed" in the rest of the program by simply selecting the proper element of the FACT array.

Consider next the binomial coefficient $\binom{n}{r}$, defined by $\frac{n!}{r!\,(n-r)!}$ for $0 \leq r \leq n$. The coefficient can be interpreted either as the number of combinations of n things taken r at a time, or as the coefficient of the term $a^r b^{n-r}$ in the binomial expansion of $(a+b)^n$. If a program needs many values of the binomial coefficients, it will be most efficient to pre-compute them. They could be stored in an array: array BINCOF [0:S, 0:S]. Here BINCOF[n,r] = $\binom{n}{r}$, for $0 \leqq r \leqq n \leqq S$. The table of binomial coefficients can be generated by the algorithm:

```
for N ← 0 step 1 until S do
    for R ← 0 step 1 until N do
        BINCOF [N,R] ← FACT [N]/ (FACT[R] * FACT[N-R] );
```

The beginning of the BINCOF array looks like:

$$\begin{array}{c|ccccc}
 & & & r \rightarrow & & \\
\hline
 & 1 & & & & \\
 & 1 & 1 & & & \\
n & 1 & 2 & 1 & & \\
\downarrow & 1 & 3 & 3 & 1 & \\
 & 1 & 4 & 6 & 4 & 1
\end{array} \qquad \text{BINCOF } [n,\ r]$$

The binomial coefficients are called "the triangular numbers" since each entry in this table is the sum of the entry above plus the entry above and to the left. Thus there is a recurrence relation among the binomial coefficients:

$$\text{BINCOF}[n,r] = \text{BINCOF}[n-1,\ r-1] + \text{BINCOF}[n-1,\ r] \ , \qquad \text{(B3)}$$

which is exploited by the following more efficient algorithm for computing the BINCOF table:

```
for N ← 0 step 1 until S do
    begin
        for R ← 1 step 1 until N-1 do          (B4)
            BINCOF[N,R] ← BINCOF[N-1,R-1] + BINCOF[N-1,R];
        BINCOF[N,0] ← 1; BINCOF[N,N] ← 1
    end ;
```

Algorithm (B4) performs only a single addition in the inner loop, instead of the multiplication and division performed by algorithm (B3). Recurrence relations used in this way often lead to more efficient programs.

Another general class of applications of arrays arises from the representation of mathematical "quantities" which involve sets of real numbers: for example, performing the operations of arithmetic on complex numbers or on polynomials.

A complex number must be represented in the computer by a pair of real numbers: the real and the imaginary parts (or the angle θ and the radius r from the polar form: $re^{i\theta}$). Thus, a complex simple variable Z must be represented as a <u>real</u> <u>array</u> Z [1:2] ; a vector of complex numbers $W_1, W_2, \ldots, W_{1000}$ must be represented by a matrix <u>real</u> <u>array</u> W[1:2, 1:1000]. In general, representation of complex numbers in the machine requires an extra dimension in every array data structure. Some algebraic compiler languages (e.g., FORTRAN IV) have complex arithmetic built in so the programmer can declare variables to be of type <u>complex</u> instead of <u>real</u> or <u>integer</u>; for example: <u>complex</u> <u>array</u> W[1:1000]. In this case, the compiler automatically adds the extra dimension to each data structure which is declared <u>complex</u>, and produces the machine language instructions to evaluate both the real and the imaginary parts of each complex variable which is assigned a value.

If $p_0, p_1, \ldots, p_n$ are n + 1 numbers (called "coefficient") and X is a dummy variable, then the formula:

$$p_0 + p_1X + p_2X^2 + \ldots + p_nX^n = \sum_{k=0}^{n} p_kX^k \qquad \text{(B5)}$$

is called a "polynomial form" of degree n. If the highest coefficient $p_n \neq 0$, then n is called the exact degree. Elementary mathematics gives rules for adding, subtracting, multiplying, and dividing such polynomial forms by suitable operations on their coefficients. Polynomial forms are often useful in numerical calculations, since many functions which are not polynomials can be approximated to any desired accuracy by suitable polynomials. The operations of arithmetic as well as the operations of calculus (differentiation and integration) can be performed very simply on polynomial forms, giving the final result as the coefficients of a new polynomial form. A numerical answer for any value of X can then be obtained from these result coefficients simply by "plugging" the value of X into the form.

A polynomial form of degree n would be represented in the computer by a vector containing its n + 1 coefficients; for example, the three polynomial forms $P_n(X)$, $Q_m(X)$, and $R_t(X)$ (of degree n, m, and t, respectively), could be represented by their coefficients in the arrays:

real array p[0:n] , q[0:m] , r[o:t] .

Suppose that $R_t(X)$ of degree (but not necessarily exact degree) t is to be computed as the sum $P_n(X) + Q_m(X)$; the algorithm would be as follows:

```
t ← if m > n then m else n;
for i ← 1 step 1 until t do                                  (B6)
    r[i] ← ( if i ≤ n then p[i] else 0 )
         + ( if i ≤ m then q[i] else 0 );
```

A polynomial form is best evaluated for a particular value X = C in the "factored form":

$$p_n(C) = (\ldots((p_n * C + p_{n-1}) * C + p_{n-2}) * C + \ldots) * C + p_o; \quad (B7)$$

The algorithm is:

```
SUM ← p[n] ;                                                 (B8)
for i ← n-1 step -1 until 0 do SUM ← SUM * C + p[i];
```

C. The Mapping Function

The internal memory of all modern digital computers is organized into cells each of which will store a numerical value. These cells are generally referred to in a program by consecutive serial numbers called addresses. For example, many large scientific computers have an internal memory of 32,768 cells, with addresses 0,1, 2,...,32767. The peculiar number "32,768" is a power of 2, because the internal representation of numbers in these machines is binary (or "base 2"). However, the particular internal number system is irrelevant to our current discussion of data structures; one needs only to understand that a memory cell contains some representation of a number.

It is convenient to picture the machine's memory as a vector of cells; then we can invent a pseudo one-dimensional ALGOL array MEM:

```
array MEM [0:32767]                                          (C1)
```

to symbolize the memory. Thus, the value stored in the cell with address 1376 can be referred to as MEM[1376], and we can use (pseudo-) ALGOL statements to describe the machine's operations on this cell.

Each machine language command typically contains (1) an "operation code", such as "ADD", or "MULTIPLY", or "STORE" (coded numerically), and (2) a single numerical memory address. We could explain the operation of each operation code in the machine using ALGOL-like assignment statements and the pseudo memory array MEM.

For example, the machine language instructions

ADD 1376 ; STO 1221 ;

might mean, in pseudo-ALGOL:

$$AC \leftarrow AC + MEM[1376] \ ; \ MEM[1221] \leftarrow AC.$$

Here the "variable" AC represents the arithmetic register (called the "ACcumulator") of the computer.

An important conclusion from the MEM array analogy is that the type of data structure intrinsic to the machine is the one-dimensional array; arrays of two or more dimensions are artificial structures built up by programming. Consider the subscripted variable $D[i,j,k,l]$; for each set of allowable values of its subscripts i,j,k, and l, there is a corresponding cell in the computer memory and therefore a corresponding element in the MEM array. This correspondence between the four subscripts and the one-dimensional MEM array is contained in the "<u>mapping function</u>" mapfct (i,j,k,l):

$D[i,j,k,l]$ is stored in $MEM[loc]$

where: loc = mapfct (i,j,k,l) .

A mapping function is generally involved in any use of an array of two or more dimensions in a computer program. Wherever a two-dimensional subscripted variable appears in an ALGOL program, for example, the ALGOL compiler inserts the machine language commands necessary to evaluate the mapping function into the object program. When the object program is executed, this mapping function must be evaluated each time the subscripted variable is referenced; thus, every reference to a subscripted variable in an ALGOL program cost some execution time. It should be emphasized, however, that this execution time is a <u>cost</u>, but not in general a <u>waste</u>; the time involved is often a negligible fraction of the entire computation, while the use of arrays generally pays large dividends in ease of programming and fewer programming errors. There are situations, however, when subscripted variables are used in the inner-most loop of a program and evaluation of the mapping functions accounts for most of the execution time; this is generally true for matrix inversion routines, for example. In this situation, the computation of mapping functions can be made more efficient at the cost of additional programming complexity, as will be described later in this section.

The mapping function for an array contains in effect both the <u>representation</u> of the array structure in memory and the <u>allocation</u> of memory space to the array. In the simplest and most common situation in programming, each array is stored in its own block of contiguous memory cells; in this case, the mapping function can be a simple arithmetic formula, as we shall see in the remainder of this section. When an array does occupy contiguous cells, the allocation of memory space involves only two numbers: the lowest memory address occupied by the array, called the "base address", and the total number of cells occupied by the array. These two quantities are calculated when the array is declared, by an algorithm which will be discussed later. For the moment, assume that a suitable allocation process has assigned base addresses to each array in the program (or, more accurately in ALGOL, to each array which is currently defined); the base address thus determined will be symbolized by the pseudo-ALGOL variable BASE.

Now let us look at the mapping functions for array data structures occupying contiguous memory space. Suppose that V is a one-dimensional array: <u>integer</u> <u>array</u> V [a:m] ; V therefore requires m -a+1 cells. If V[a] is stored in MEM[BASE] and if V is stored in consecutive cells, then V[i] is stored in MEM[BASE + i-a] for i = a, a+1,..., m. That is, the mapping function for one-dimensional arrays is simply additive. The sum (BASE - a) can be computed and stored when the array is declared, so only one addition need be performed when the array is accessed.

Two-dimensional arrays are generally stored in a computer in order either by row* or by column** because this leads to the simplest form of the mapping function. For example, consider an array: <u>array</u> A[1:m, 1:n]. If A is in order by row, the elements of A would be stored in the order: A[1,1],...,A[1,n], A[2,1],...,A[2,n], ..., A[m,1],..., A[m,n]. Again, assume that A[1,1] is stored in MEM[BASE] and successive elements are stored in consecutive memory cells; then it is easy to see that:

A[i,j] is stored in MEM[BASE + (i-1)*n + j-1] .

Notice that n is the number of columns in the array. The mapping function is given by:

$$\text{mapfct}\ (i,j) = \text{BASE} + (i-1)*n + j-1 \qquad (C2)$$

The declaration of the array determines the two values: m and (BASE-n-1), so that evaluation of the mapping function to access A[i,j] requires one multiplication and two additions.

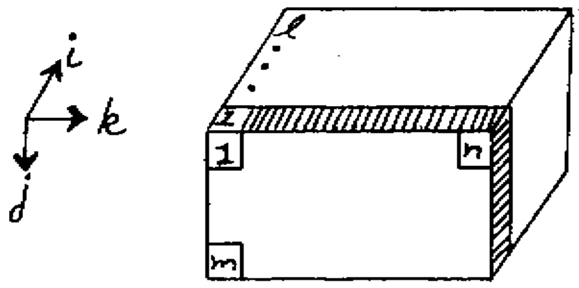
The mapping function for a two-dimensional array can be applied to accessing a three-dimensional array. Assume that:

---

* As in ALGOL-20
** As in FORTRAN on an IBM 7090

array $C[1:\ell,\ 1:m,\ 1:n]$.

This can be pictured as a rectangular solid figure:



Let the first subscript vary most slowly as we move through successive addresses; i.e., first comes $C[1,j,k]$ for all j and k, then $C[2,\ j,k]$, etc., until $C[\ell,\ j,k]$. Thus each value of the first subscript i of $C[i,j,k]$ fixes a particular (j,k) plane, which is a two-dimensional array whose mapping function is given by Eq. (C2). Since each plane contains m*n elements, the three-dimensional mapping function can be written:

$$\text{mapfct}\ (i,j,k) \Rightarrow \text{BASE} + (i-1) * m*n + (j-1)* n + k-1$$

This can be factored into the form:

$$\text{mapfct}\ (i,j,k) = \text{BASE} + ((i-1)* m + j-1)* n + k-1 \qquad \text{(C3)}$$

This expression illustrates the general rule for construction of a mapping function for an n-dimensional array in "row-order":

$$\text{mapfct}\ (i,j,\ldots p,q) = \text{BASE} + ((\ldots((i-1)* \delta_2 + j-1)* \delta_3 + \ldots)* \delta_{n-1} + p-1)* \delta_n + q-1 \qquad \text{(C4)}$$

for an array: **array** $G[1:\delta_1,\ 1:\delta_2,\ldots,\ 1:\delta_n]$.

In all these mapping functions, we have used 1 as the lower bound for all subscripts. The lower bound could be any number, changing the mapping function in an obvious way. In the general case, suppose:

array $H[\ \lambda_1:\ \mu_1\ ,\ \lambda_2:\mu_2\ ,\ \ldots,\lambda_n:\mu_n\ ]$

so that the lower bounds are $\lambda_1,\ldots\ \lambda_n$ and the upper bounds are $\mu_1,\ldots\ \mu_n$. Then the "dimensions" $\delta_1,\ldots,\ \delta_n$ are given by:

$$\delta_1 = \mu_1 - \lambda_1 + 1,\ \delta_2 = \mu_2 - \lambda_2 + 1,\ldots,\ \delta_n = \mu_n - \lambda_n + 1\ , \qquad \text{(C5)}$$

and the mapping function is:

$$\text{mapfct } (i,j,\ldots,p,q) = \text{BASE} + ((\ldots((i - \lambda_1) * \delta_2 + j - \lambda_2) * \delta_3 + \ldots) * \delta_{n-1} + p - \lambda_{n-1}) * \delta_n + q - \lambda_n. \qquad \text{(C6)}$$

This formula can be rewritten as:

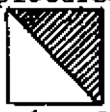
$$\text{mapfct } (i,j,\ldots,p,q) = \text{BASE} + ((\ldots(i*\delta_2 + j) * \delta_3 + \ldots)* \delta_{n-1} + p) * \delta_n + q + C \qquad \text{(C7)}$$

where

$$C = ((\ldots((-\lambda_1)* \delta_2 - \lambda_2)* \delta_3 + \ldots)* \delta_{n-1} - \lambda_{n-1})* \delta_n - \lambda_n \text{ .}$$

Since C depends only on λ and δ, the quantity (BASE + C) need be evaluated only once when the array is declared. Then evaluation of (C6) requires (n-1) multiplications and n additions.

More complicated mapping functions are possible, and sometimes useful, where the added execution time to evaluate them is more than compensated by the saving in memory space. For example, it sometimes happens (e.g. the BINCOF array in section B) that we don't need all the elements A[i,j] of a matrix, but need only those on or below the diagonal; that is, we need the elements which satisfy the relation: $0 \le j \le i \le n$. This can be pictured as the unshaded area in:



and is called the lower triangle of the matrix. We might wish each row of the matrix in memory to be one element longer than the preceding row so as not to waste any memory space. Thus, successive memory cells would contain the $\frac{n\ (\ n + 1)}{2}$ elements:

$$\begin{array}{l} a_{11}, \\ a_{21}, a_{22}, \\ a_{31}, a_{32}, a_{33}, \\ \vdots \qquad\quad \vdots \\ a_{n1}, a_{n2}, a_{n3}, \ldots, a_{nn} \end{array}$$

The mapping function for array A[1:m, 1:n] would be given by:

$$\text{mapfct } (i,j) = \text{BASE} + \frac{i(i - 1)}{2} + j - 1 \ ; \qquad \text{(C8)}$$

(see Section 6.7 in McCracken's text). Such a mapping function is not built into the ALGOL translator (although it could be if there were sufficient need for it); therefore, to use it the programmer must write the mapping expression explicitly himself. For example, he could declare an array L[1: $\frac{n(n+1)}{2}$] (i.e., a vector) instead of the array A; whenever he wanted A[i,j] in his program he would use:

L[(i * (i-1))/2 + j] .

We have seen that the mapping function for an n-dimensional array stored in consecutive memory cells requires n-1 multiplications. On most machines, unfortunately, multiplication is relatively costly in time. To speed the evaluation of mapping functions, some compiler systems* use subsidiary data structures called access tables, which are the subject of Section D. We will find that access tables not only save multiplications, but are in fact much more flexible than the simple arithmetic functions such as Eq. (C4); they will give us a convenient representation for more general abstract structures, and will allow us to relax somewhat the requirement of strictly contiguous memory space for an entire array.

Before we consider access tables, however, we should observe that there is another approach to program efficiency in the evaluation of array mapping functions: partial pre-evaluation. This is nothing more than an application of the simple principle of efficiency that any part of a calculation that doesn't change within a loop should be performed once, outside the loop. Suppose that the variable A[i,j] is to be accessed within an inner-most loop of a program and that this loop is stepping through values of j with i fixed; then the quantity: BASE + (i-1)* n - 1 could be evaluated outside the loop, and the single addition of j would complete the evaluation inside the loop.

For example, we could apply this technique explicitly to make the binomial coefficient program (B4) more efficient. Each time the binomial recurrence relation there is evaluated, there are three multiplications to be performed, hidden in the mapping function of the two-dimensional BINCOF array. If algorithm (B4) is executed many times so that utmost efficiency is important, the programmer can save these multiplications by explicit pre-evaluation of the mapping function. Thus, the programmer could define a one-dimensional array:

* For example, the ALGOL-20 compiler at Carnegie Tech.

array BC1D [0: $(s+1)^2-1$], where BINCOF[n,r] corresponds to BC1D [(s+1)*n + r]. The algorithm would then become:

```
C1 ← (s+1) + 1 ; C2 ← (s+1) ;
for n ← 0 step 1 until s do                                    (C9)
     begin p ← n*(s+1);
     for r ← 0 step 1 until n do
          begin SUBS ← p + r;
          BC1D[SUBS] ← BC1D[SUBS-C1] + BC1D[SUBS-C2]
          end ;
     end ;
```

Notice that: (1) the constants C1 and C2 are computed only once, and (2) the single multiplication n*(s+1) in the mapping function is independent of r and has therefore been moved out of the inner-most loop.

In making this change, the programmer has gained efficiency but has given up the simplicity and clarity of the matrix notation; he is liable to have more trouble debugging an algorithm like (C9) then one like (B4). Fortunately, some compilers automatically check for invariant calculations and move them outside inner loops, and would compile an object program which uses the algorithm (C9), given the source program (B4). Using one of these optimizing compilers, a programmer can have an efficient program without giving up the simplicity of the general array notation. There is an associated cost, of course; an optimizing compiler must make a complex analysis of the source program to determine which parts of the calculation are invariant, and therefore such a compiler will be significantly slower than a compiler which performs no optimization but simply produces an object program which re-evaluates the complete mapping function for each array access.

An optimizing compiler will generally perform another type of optimization of mapping function evaluation to take advantage of machine registers called "index registers". We will not discuss this further, except to note that this use of index registers is an application of the general array mapping function, Eq. (C4).

D. Access Tables

To introduce access tables, we will again consider two-dimensional arrays in contiguous storage space. We found earlier that the function for array A[1:m, 1:n] was given by the rule:

$$A[i,j] \text{ is stored in } MEM[BASE_A + (i-1)* n + j - 1].$$

Suppose we declare an auxiliary array: array A1[1:m], and fill it with pre-computed values of $(BASE_A + (i-1)* n - 1$.

$$A1[i] = BASE_A + (i-1)* n-1 \quad \text{for } i = 1, 2, \ldots, m \tag{D1}$$

The A[I,j] is stored in $MEM[MEM[(BASE_{A1} - 1) +i] +j]$ so the array mapping function (C2) can be evaluated by:

$$mapfct\ (i, j) = MEM[(BASE_{A1} - 1) + i] + j, \tag{D2}$$

which requires no multiplications. The A1 array is called an access table, or access vector; it contains one word for each row of the matrix A.

Consider next a three-dimensional array: array $B[1:\ell, 1:m, 1:n]$. Applying the access table idea, we can create a two-dimensional access table array $B1[1:\ell, 1:m]$, and fill it with pre-computed values: (D3)

$$B1[i,j] = BASE_B + ((i-1)* m + j-1)*n, \text{ for } i = 1,\ldots,\ell \text{ and } j = 1,\ldots,m .$$

But B1 is itself a two-dimensional array whose mapping function can be evaluated with the aid of an access vector array $B2[1:\ell]$, where

$$B2[i] = BASE_{B1} + (i-1)* m - 1, \text{ for } i = 1,\ldots, \ell . \tag{D4}$$

Putting all these together, we get a mapping function for B[i,j,k] which requires no multiplications:

$$mapfct\ (i,j,k) = MEM[MEM[(BASE_{B2} - 1) +i] +j] + k \tag{D5}$$

Clearly the same principle can be applied to accessing arrays with any number of dimensions. An n-dimensional array requires n-1 access tables with

$$\delta_1, (\delta_1 x\delta_2), (\delta_1 x\delta_2\delta_3),\ldots,(\delta_1 x\delta_2 x\delta_3 x\ldots x\delta_{n-1}) \tag{D6}$$

elements, respectively; the sum of these terms gives the total memory space required by the access tables.

Notice that the use of access tables saves execution time, but the tables require their own memory space. Thus, the decision to use access tables for subscripted variables involves a trade-off which is frequently possible in programming: execution time vs. memory space. In general, however, the cost in memory space for an access table is small. A square n x n matrix and its access table require n(n+1) cells, and $n^2 < n(n+1) < (n+1)^2$. Thus, sufficient space for an access vector can always be obtained merely by reducing the maximum size of a square matrix by one. A similar argument applies to arrays with more dimensions - the space for access tables at worst reduces the possible range of each subscript position by one (assuming that the array is approximately NxNx...xN). The situation is somewhat different if one or more subscripts have a much larger range than the others. For example, the access vector for array C[1:2, 1:1000] requires 2 extra cells, while the access vector for array C[1:1000, 1:2], with the rows and columns interchanged, requires 1000 extra cells. Therefore, if one is short of memory space for data and has such a "long, thin" array, it is best to use the "thin" dimension in the first subscript position.

It is important to realize that an ALGOL system* which uses access tables for array mapping functions does so completely automatically; the user need consider these implicit access tables only if he needs to account for the memory space they use. When an n-dimensional array is declared, the system automatically declares the corresponding n-1 access tables and generates the proper integers in these tables. This task will be performed by a run-time subroutine which we will call "RAD" for Run-time Array Declarations; on the next page there is a pseudo-ALGOL version of RAD. When an array declaration contains a list of M identifiers with one common set of bounds:

$$\langle\text{identifier}_1\rangle, \ldots \langle\text{identifier}_M\rangle\ [\ \langle\text{lower bound}_1\rangle : \langle\text{upper bound}_1\rangle, \ldots, \langle\text{lower bound}_N\rangle : \langle\text{upper bound}_N\rangle\ ]$$

then RAD creates only one set of access tables common to all M arrays. The first array, $\langle\text{identifier}_1\rangle$, in the set is accessed the normal way; any other array in the set is accessed by evaluating the common mapping function (giving the mapping for $\langle\text{identifier}_1\rangle$ and then adding an origin shift or "offset" term to get the final memory address in the desired array.

One of the functions of RAD is to reserve (or "allocate") memory cells for the access tables and arrays; the function of LEVEL and AVAIL in this allocation process can be fully explained only when ALGOL block structure has been covered.**

---

* (e.g. The ALGOL-20 system at Carnegie Tech).

** See, for example:
Sattley, K., "Allocation of Storage for Arrays in ALGOL 60", Comm. Assoc. Comp. Machinery, 4 (January 1961), pp. 60-64.

```
Comment ALGOL-20 Run-Time Array Declaration Routine, in pseudo-ALGOL;
     begin
          array MEM[0:32767], AVAIL[1:64] ; integer LEVEL;
               comment  MEM is pseudo-array representing G-20 memory,
                    AVAIL[LEVEL] is first available location in array
                       storage in current block;
          procedure RAD (DBLPREC, M, N, λ, μ, DOPE) ;
               Boolean DBLPREC; integer M, N ;
               integer array λ, μ, DOPE ;
                    comment
                         DBLPREC = true if array is of type real, false
                                   otherwise,
                         M = Number of arrays declared together,
                         N = Number of dimensions,
                         λ[1:N] = Vector of lower bounds,
                         μ[1:N] = Vector of upper bounds,
                         DOPE[1:M, 1:4] = Set of M 4-element "dope vectors",
                                          to be computed;
                    begin
                         integer array δ, SIZE[1:N];
                         integer BASE, I, J, K, PROD, SPACE, VO ;
                         PROD ← 1; SPACE ← 0 ;
                         for K ← 1  step  1  until  N  do
                              begin
                                   δ[K] ← μ[K] - λ[K] + 1  ;
                                   SIZE[K] ← PROD ← PROD * δ[K]  ;
                                   if K < N then SPACE ← SPACE + PROD
                              end ;
                         comment  Now SPACE is total memory space required
                                  for all of the N-1 access tables;
                         if  DBLPREC  then
          begin
               SIZE[N] ← 2 * SIZE[N] ;
               δ[N] ← 2 * δ[N] ; λ[N] ← 2 * λ[N]
          end adjustment for double precision;
     SPACE ← SPACE + M * SIZE[N] ;
          comment SPACE is now the total number of
                  memory cells for M arrays and their
                  Access Tables;
     AVAIL[LEVEL] ← AVAIL[LEVEL] - SPACE ;
     VO ← BASE ← AVAIL[LEVEL] + 1 ;
          comment VO = the address of the first cell of
                       Access Table ;
     comment Now build up the N-1 access tables;
     for K ← 1 step 1 until N-1 do
          begin
               comment Access Table K has its first
                       entry in MEM[BASE] ;
               MEM[BASE] ← BASE + SIZE[K] - λ[K + 1];
               for I ← 1 step 1 until SIZE[K] - 1 do
                    MEM[I + BASE] ← MEM[I + BASE - 1]
                       + δ [K + 1] ;
               BASE ← BASE + SIZE[K]
          end generation of Access Table K ;
     comment Now set Dope Vectors ;
for J ← 1 step 1 until M do
     begin
          DOPE[J,1] ← VO - λ[1] ;
               comment This is origin address for
                       the mapping function;
          DOPE[J,2] ← (J-1) * SIZE[N] ;
               comment This is final "offset" for
                       the Jth array;
                    DOPE[J,3] ← BASE + DOPE[J,2] - 1 ;
                        comment  This is a lower limit of
                                 array, for bound check;
                    DOPE[J,4] ← DOPE[J,3] + SIZE[N] ;
                        comment  This is upper limit of
                                 array, for bound check;
                end
            end  RAD;
```

comment s:

1. Example: The declaration: integer array A,B,C,[1:10, -6:8] compiles a call of RAD of the form:

   RAD(false, 3, 2, λ, μ, DOPE)

   where:

   $\lambda[1] = 1, \quad \lambda[2] = -6$

   $\mu[1] = 10, \quad \mu[2] = 8.$

   If AVAIL[LEVEL] = 20,000 before RAD is called, the result will be to compute the 3 DOPE vectors:

   ```
   DOPE[1,...] = 19540,   0, 19550, 19700.
   DOPE[2,...] = 19540, 150, 19700, 19850.
   DOPE[3,...] = 19540, 300, 19850, 20000.
   ```

2. General Mapping for L^{th} array out of set of M array identifiers declared together:

   $$\text{mapfct}(\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n) = \text{MEM}[\text{MEM}[\ldots\text{MEM}[\text{MEM}[\text{DOPE}[L,1] + \mathcal{E}_1] + \mathcal{E}_2]\ldots] + \mathcal{E}_{n-1}] + (\underline{\text{if}}\ \text{DBLPREC}\ \underline{\text{then}}\ 2 * (\mathcal{E}_n)\ \underline{\text{else}}\ \mathcal{E}_n) + \text{DOPE}[L,2];$$

end of pseudo-ALGOL RAD Explanation

We have described the manner in which some ALGOL systems use access tables for the efficient evaluation of the normal subscript mapping functions, Eq. (C4). The programmer who needs a more complex or more general mapping function than Eq. (C4) can often achieve considerably greater efficiency and simplicity in his program by using access tables which he has explicitly created for his purpose. If he is writing his program directly in a machine-language representation (that is, in "assembly" language), he can create and use access tables directly in the MEM "array", as we have been describing. If he is writing in ALGOL, however, he will need to declare one-dimensional ALGOL arrays for his data array and access tables; as we have seen, a one-dimensional array is related to the MEM array by a very simple additive mapping function: essentially, the BASE address is added to the subscript.

For example, refer again to the lower triangular array considered on page 44. Suppose we declare a vector Q which will contain the information which would have been in the A matrix, and also an access vector AT:

```
array Q[1: (N * (N + 1)/2)]; integer array AT[1:N];
```

At the beginning of execution we will execute a loop to fill AT with suitable values:

```
AT[1] ← 0; for I ← 2 step 1 until N do AT[1] ←             (D7)
              I + AT[I-1] - 1;
```

Then: $AT[i] = \frac{i(i-1)}{2}$, for $i = 1,2,\ldots,N$. Wherever A[I,J] would have appeared in the program, we use the access table AT explicitly, in the form:

$$Q\ [AT\ [I] + J]. \tag{D8}$$

As another example of the general use of access tables, suppose we interchange two single entries say, A1[i] and A1[j], in the access table A1 of Eq. (D1). A little thought will show that the effect is to interchange the i^{th} row with the j^{th} row, without physically moving any of the data words in the matrix A. Thus, we have an efficient way of interchanging entire rows of a matrix at once, which is an important problem in programming. For example, several good algorithms for solving large sets of simultaneous linear equations require interchanging entire rows of the matrix of coefficients to prevent excessive round-off error. Another example is provided by data processing tasks which require sorting n items, each consisting of m words. Each multiword item can conveniently be represented as a single row of an m-column array DATA[1:n, 1:m]. Sorting these m-word items may require interchanging pairs of items - but since each item is an entire row of m words, this can be costly in time. Instead, we set up an access vector, each element of which "points to" the beginning of the corresponding item in memory; then the items can be sorted by reordering only the pointer words in the access vector.

Access tables can provide considerable flexibility in allocation of space in the computer memory. The mapping functions for multidimensional arrays discussed in Section C assumed that an array was stored in "row-order" in one contiguous block of memory cells. The idea of interchanging rows of a matrix by interchanging access vector elements suggests that if access tables are used the rows could be in arbitrary order and could be placed anywhere in memory. Each complete row must, of course, still have contiguous storage assigned to it.

It is even possible that some or all of the rows of a matrix may not have any core storage allocated to them at all - these rows may be on a magnetic tape or magnetic disc file. Many real-life problems require a great deal more data storage than is available in the machine's high-speed memory. When a program requires a matrix which is too large to fit entirely into high-speed memory, one solution is to allocate high-speed memory space only for one or a few rows which are currently being used. The elements of the access vector corresponding to the rows currently in memory would, of course, point to these rows; all the other access vector elements, corresponding to rows currently on a magnetic tape (for example) would contain an "interrupt flag". When the ALGOL program attempted to access a row which was not in memory, the machine's circuitry would detect the interrupt flag on the access vector element and automatically take special action; the result would be to interrupt the program in progress long enough to execute an administrative program which would read the missing row into high-speed memory, allocate space to it (writing another row out, if necessary, to make room), adjust the access vector to reflect the new state of affairs, and then finally return to the suspended program and complete the array access.

E. Jagged Arrays, Trees, and Lists

In Sections C and D we considered the memory representation of arrays of subscripted variables which have a very simple abstract structure; we will now discuss some more complex and general abstract data structures.

We will begin by generalizing array structure in a very simple way: we will allow each row of a matrix to have a different length, each plane of a three-dimensional array to have a different number of rows, etc. This generalization is easy to implement if access tables are used for the mapping function; with suitable changes in the contents of the access tables, the mapping functions of Eq. (D2) and Eq. (D5) will work perfectly for these arrays with "jagged edges" To see this, it is helpful to think of the mapping function (D5) in terms of the following picture:

There is no way to declare a jagged array in ALGOL-60 -- we would need for clauses within the array declarations, in some form such as:

$$\underline{\text{integer}}\ \underline{\text{array}}\ \text{TEXT}[\underline{\text{for}}\ S \leftarrow 1\ \underline{\text{step}}\ 1\ \underline{\text{until}}\ \text{NSENT}\ \underline{\text{do}}\ (S, \underline{\text{for}}\ W \leftarrow 1\ \underline{\text{step}}\ 1\ \underline{\text{until}}\ \text{NW}[S]\ \underline{\text{do}}\ (W, 1{:}\text{NL}[W,S]))]; \quad \text{(E3)}$$

On the other hand, the programmer can explicitly build his own access tables for TEXT in a manner analogous to the lower triangle mapping function of Eq. (D8).

Access tables lead naturally to a much more significant change in abstract structure than a jagged array. The accessing operation, shown pictorially in (E1) for the case of a three-dimensional array, generally goes through n-1 levels of access tables for an n-dimensional array. Suppose that for some values of the subscripts we cut short the accessing process at fewer than n levels, so that the element of an access table at which we terminate contains the data value we want, rather than a pointer to the next level of accessing. Thus, we remove the rigid distinction between an access table and the final data array; an access table will now contain a mixture of data words, and pointers to deeper levels of subscripting. Such a structure is called a tree.

We have just described the memory representation of a tree, before giving it abstract structure. To describe its abstract structure, we need a new notation. Consider first the simple array structure array A[1:2, 1:3]; its abstract structure could be described by listing all its elements and using parentheses to group the elements of a row (or in the general case, the rows of a plane, the planes of a..., etc.), as follows:

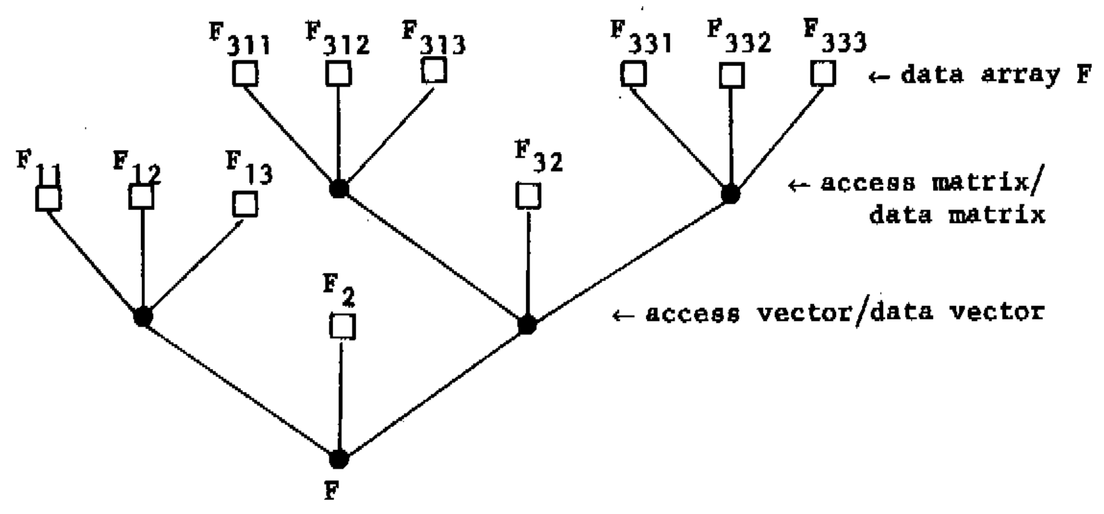
$$(a_{11}, a_{12}, a_{13}), (a_{21}, a_{22}, a_{23}). \quad \text{(E4)}$$

In this representation, each left parenthesis corresponds to one level of addressing, i.e., following a pointer from one access table to another. In the case of simple ALGOL array structures and simple jagged arrays with n dimensions, each data element a_{ij} will lie within n-1 pairs of parentheses. The generalization to tree structures simply removes this restriction on the number of parentheses. For example, we could have the following abstract tree structure F: (E5)

$$(F_{11}, F_{12}, F_{13}), F_2, ((F_{311}, F_{312}, F_{313}), F_{32}, (F_{331}, F_{332}, F_{333})).$$

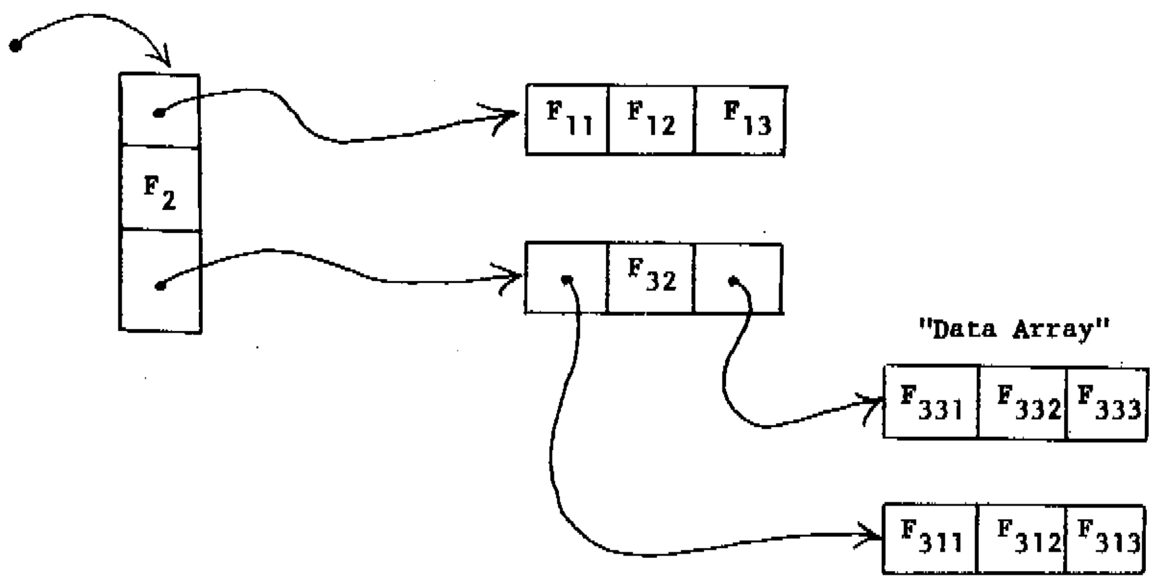
The reason this is called a tree is that it can conveniently be visualized as the following tree-like structure:

(E6)



Here the square nodes signify data words, while the dots signify pointers to the next level of accessing. The method of accessing this tree structure can be shown in a picture analogous to (E1).

(E7)



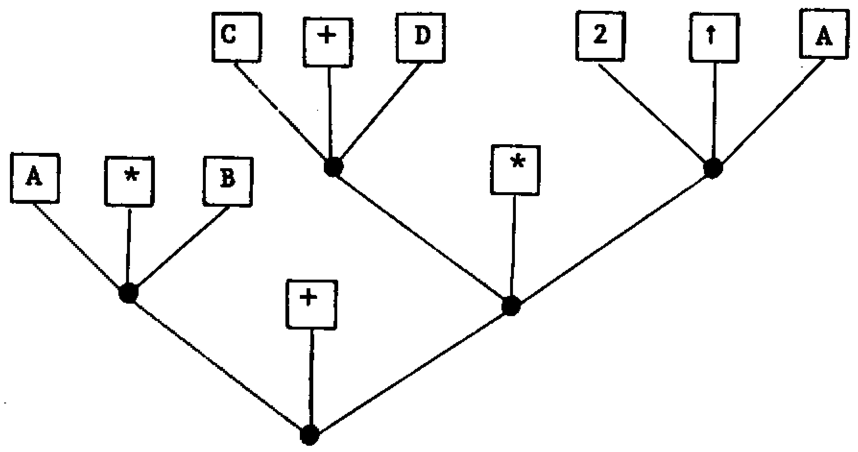
We can define the abstract structure of a tree in the following way: a tree is a list (or vector) of terms, each of which may be either an elementary data value or else another tree.

Finally we will give a particular example of the tree structure of (E5) and (E6) , a representation of the ALGOL arithmetic expression:

$$A * B + (C + D) * 2\uparrow A \qquad \text{(E8)}$$

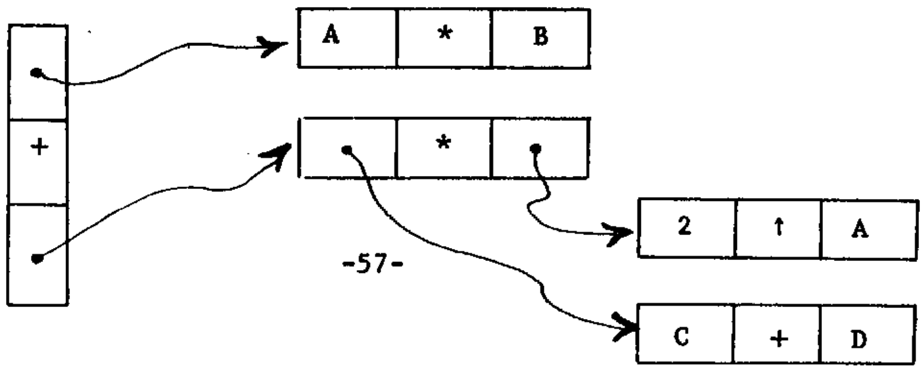
Applying the precedence rules, we find that both the sum (C+D) and the power 2↑A must be evaluated before the product (C+D)*2↑A, and both products must be performed before the final sum. This structure is represented in a natural manner by the following tree, which has an abstract structure identical to (E6).

(E9)



"Growing from" each dotted node are an operator and two operand branches; the operation at this node can be performed only after all operations above it along either of its operand branches have been performed; when the operation has been completed at the node marked "Expression", the entire expression has been evaluated. The representation of this expression in memory looks as follows:

(E10)



Tree structures are fundamental to many complex information processing problems including, forexample, language translation, mechanical theorem proving, and chess playing programs. These problems are generally characterized by very large and complex data structures which grow, shrink, and change many times during the calculation. The result is that dynamic allocation of memory space to the data structure becomes the central problem. The usual solution to the allocation problem employs "binary branching" tree structures which have exactly two branches growing from each node; it can be seen that this makes every "access table" exactly two words long. The resulting representation is called a <u>list</u>; there are several <u>list processing languages</u>, including IPL and LISP, which use this list structure exclusively.

F. Exercise on Data Structures

<u>Exercise 1</u>:

A physicist wrote an ALGOL program to compute a 7 x 13 array of numbers C_{LM} where:

$$0 \leqq L \leqq 6$$

$$-6 \leqq M \leqq +6$$

Furthermore, the C_{LM}'s were always zero unless M satisfied

$$-L \leqq M \leqq L,$$

so he stored the 49 non-zero C_{LM}'s compactly in elements Q[1],...,Q[49] of a one-dimensional array Q. The C_{LM}'s were mapped into Q in the order:

$$C_{00}, C_{1-1}, C_{10}, C_{11}, C_{2-2}, C_{2-1}, C_{20}, C_{21}, C_{22}, \cdots , C_{66} ,$$

with $Q[1] = C_{00}$ and $Q[49] = C_{66}$.

To evaluate the mapping function, he defined his own access vector INDEX[0:6] and stored C_{LM} in Q[INDEX[L] + M]. What values were needed in INDEX?

<u>Exercise 2</u>:

Assume that A is an N x N array such that A[i,j] = 0 when $|i-j| > K$; such an array is called (2K + 1) - diagonal. The N + 2NK - K(K+1) non-zero elements of A are to be stored compactly in an array Q and accessed viz an access vector AT as in equation (D8). Determine the values which must be in the access vector AT.

Exercise 3:

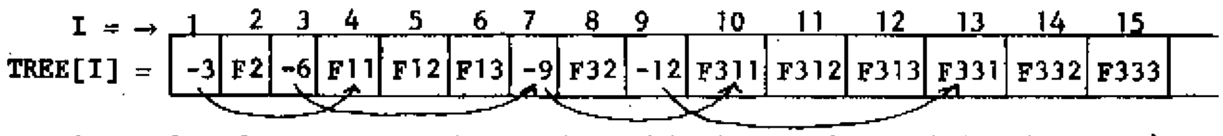
Assume that you have been given a tree structure, represented by mixed access table/data arrays as described in Section E above, in an integer array TREE[1:500] (analogous to the MEM array). A particular word TREE[I] contains the following:

if TREE[I] $\geq$ 0 then TREE[I] = a data value;

if TREE[I] $<$ 0 then -TREE[I] = an access pointer - i.e.,

TREE[-TREE[I] + 1] is the first word of the access/data table for the next access level.

Assume that subscripts always run from 1 up, so that an access pointer is actually the address one before the address at which it "points". For example, the F array, (E7) in the notes, might appear in TREE as follows:



I = →
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
TREE[I] =
-3 F2 -6 F11 F12 F13 -9 F32 -12 F311 F312 F313 F331 F332 F333

where the F's represent data values (which must be positive integers).

Consider an element of the F array with L subscripts having the particular values e_1, e_2, ... , e_L; this element would be stored in TREE[m], where:

$$m = -\text{TREE}\ [\dots\ -\text{TREE}[\ -\text{TREE}[e_1] + e_2]\ \dots\ ] + e_L.$$

Here L, the number of subscripts used, is called the level of accessing for m. It is useful to compute m by the above access formula for values of L less than the level of the corresponding data term; the result would be a (negative) pointer to the next access level. On the other hand, it is meaningless to evaluate m for L greater than the level of the data term.

You are to write a declaration for an ALGOL procedure TREE.MAP which, given subscript values e_1, e_2, ... , e_L and a value for the level L, evaluates the mapping formula above for m. The result for m is to be the value of the integer function designator TREE.MAP.

Assume:

integer procedure TREEMAP (TREE, E, L) ;

where:

TREE = integer array containing tree being accessed,
E = integer array containing subscript values:
E[1] = e_1, E[2] = e_2, etc.

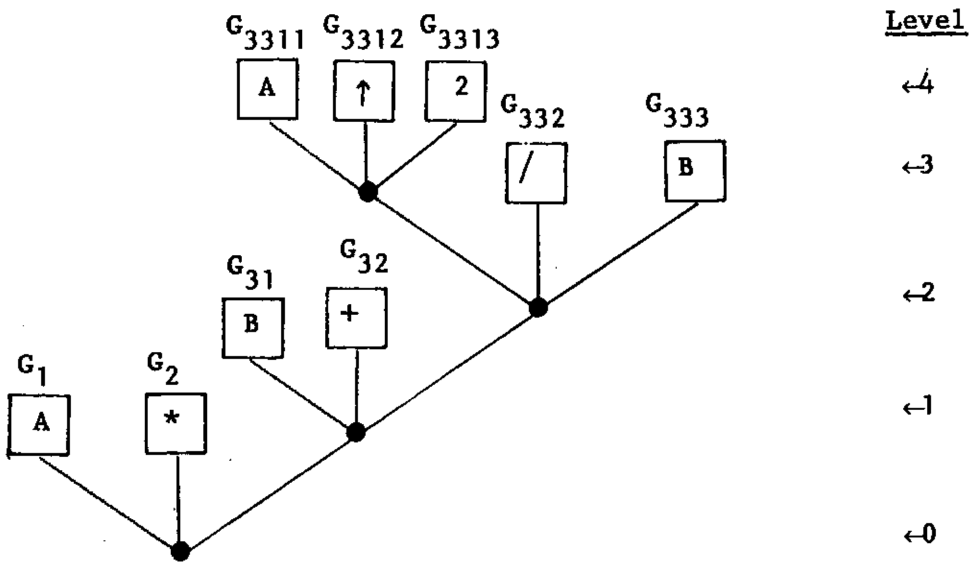
L = integer simple variable, called by name

If a data term (i.e., a non-negative value in TREE...) is encountered at a level lower than the value of L when TREE.MAP is called, then TREE.MAP should change the given value of L to the level of the data term and TREE.MAP should have as value the m for the new L (i.e., TREE[m] should be the data term).

Notice that L is used both as an input and as an output parameter. Since TREE.MAP is a function designator, it is used as an operand in an arithmetic expression; calling TREE.MAP, however, may change the actual parameter substituted for L; this is called a "side-effect" of the procedure call.

Exercise 4:

A good example of data having the abstract structure of a tree is provided by ALGOL arithmetic expressions, considered as strings of symbols. Each operator has one or two operands, each of which in turn can by any arbitrary arithmetic expression; this hierarchical structure is typical of a tree, each branch of which can be the "trunk" of an entire subtree. The tree representation of a symbolic arithmetic expression shows each complete operand of each operator, without any parentheses; therefore, the tree form is called "parentheses-free" (although a tree is not the only parentheses-free form for an arithmetic expression). For example, the expression $A * (B + A \uparrow 2/B)$ has the tree representation:



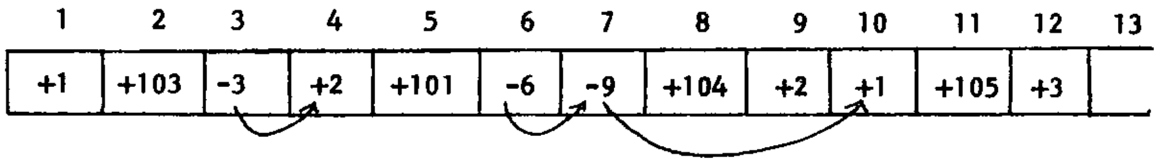
This diagram also shows the data terms labeled as subscripted variables.

In this problem, you are given a tree representation of a symbolic arithmetic expression in an integer array TREE, as well as a real array VAL which contains a specific value for each variable appearing in the expression. You are to write a program which evaluates the given expression, using the given set of values. The tree is represented in TREE in the form discussed in Exercise 3. The positive integer data values are coded as follows to represent variables and operators:

A value K in TREE[I], where K > 0, represents:

$1 \le K \le 100$:	a variable or constant whose value is stored in VAL[K];
K = 101:	operator (binary) +
K = 102:	operator (binary) -
K = 103:	operator *
K = 104:	operator /
K = 105:	operator ↑

Tree[1] will be the "trunk" of the expression tree. For example, the expression A *(B + A↑2/B) might look as follows



1	2	3	4	5	6	7	8	9	10	11	12	13
+1	+103	-3	+2	+101	-6	-9	+104	+2	+1	+105	+3	

if VAL[1] = A, VAL[2] = B, and VAL[3] = 2.

We suggest the following algorithm for evaluating the expression. Use the TREE.MAP procedure written in Exercise 3 to step through all data terms of the tree in order, as if one were stepping through all elements of an array. TREE.MAP tells the level of each term, indicating which subscript to step next. When values are known both operands of a particular operator - i.e., both operands are variables or have been evaluated previously - perform the operation. Then place the new value in an unused cell in the VAL array, and insert a new variable corresponding to this element of VAL into TREE, replacing the access pointer which was the root of the subtree you have now evaluated.

Example:

The expression A $*$(B + A↑2/B) used earlier would be evaluated in the following steps:

	"Array Element" accessed	Value	Action
1.	G_1	(A)	none
2.	G_2	*	none
3.	G_{31}	(B)	none
4.	G_{32}	+	none
5.	G_{3311}	(A)	none
6.	G_{3312}	↑	none
7.	G_{3313}	2	Evaluate T_1 = A↑2; put it in VAL and make G_{331} a pointer to it.
8.	G_{331}	(T_1)	none
9.	G_{332}	/	none
10.	G_{333}	(B)	Evaluate $T_2 = T_1/B$, put it in VAL, and make G_{33} a pointer to it.
11.	G_{31}	(B)	none
12.	G_{32}	+	none
13.	G_{33}	(T_2)	Evaluate $T_3 = B + T_2$, put into VAL, and make G_3 a pointer to it.
14.	G_1	(A)	none
15.	G_2	*	none
16.	G_3	(T_3)	Evaluate $A*T_3$. Finished.

<u>Exercise 5</u>:

Write an ALGOL program which, given a value K and the tree representation of an arithmetic expression (as in Exercise 3), determines whether the expression is an even function, an odd function, or neither, of the variable X_K whose value is stored in VAL[K]. Treat all other variables as constants whose values are given in the VAL array.

Note: a function f(X) is:

an even function if $f(-X) = f(X)$,

an odd function if $f(-X) = -f(X)$.

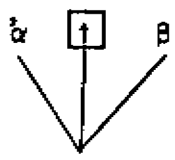
We suggest an algorithm very similar to that used in Exercise 4. Using an array parallel to TREE, put a three-valued flag at each node of the expression to indicate whether the complete sub-expression sprouting from that node is even, odd, or neither. Sequence through the nodes just as in Exercise 4.

Exercise 6:

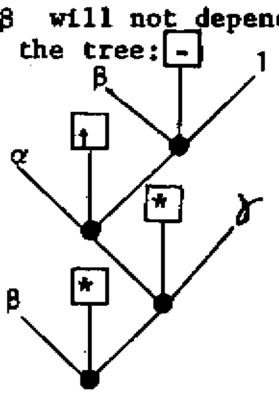
Write a program which, given K and the tree representation of an expression, produces a new tree which is the partial derivative of the given expression with respect to the variable X_K stored in VAL[K]. Thus:

$$\frac{\partial}{\partial X_K}\ \text{VAL[K]} = 1\ ;\ \frac{\partial}{\partial X_K}\ \text{VAL[I]} = 0 \text{ for all } I \neq K.$$

Assume that at every node containing an $\uparrow$ operator between sub-expressions α and β

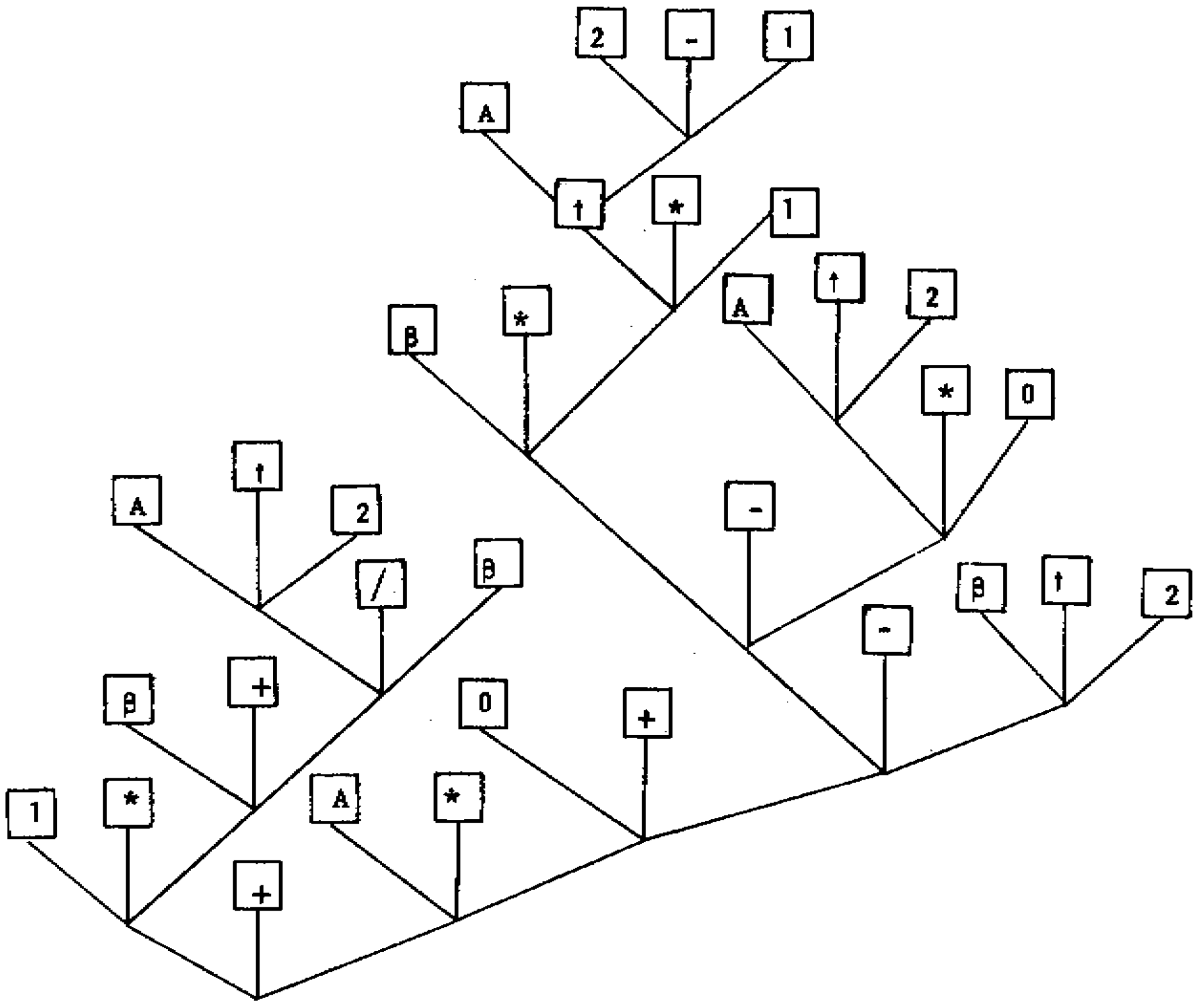


the exponent expression β will not depend upon X_K: hence the derivative of this node will be the tree:



where γ is the derivative of the subtree α .

Do not try to simplify the resulting tree by combining constants or eliminating products with 1, sums with 0, etc. For example, if the tree shown in Exercise 4 for the expression $A*(B + A^2/B)$ is differentiated with respect to A (i.e., K = 1), the result should be:

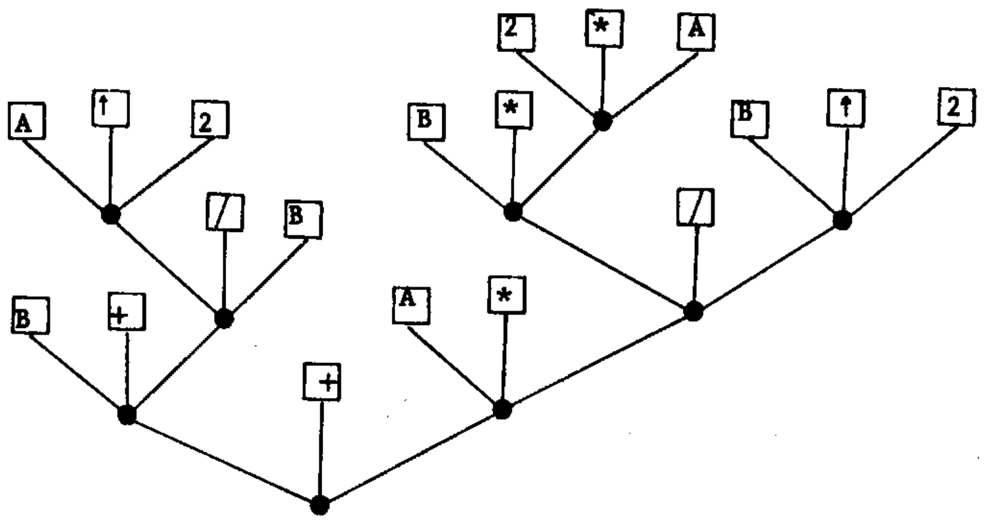


Exercise 7:

Write a program which, given K and a tree produced by the differentiation program of Exercise 6, simplifies it by applying the following algebraic rules:

(1) $0 * A = 0, \quad A * 0 = 0$
(2) $1 * A = A, \quad A * 1 = A$
(3) $0 + A = A, \quad A + 0 = A$
(4) $A \uparrow 1 = A$
(5) $A \uparrow 0 = 1$

Also, evaluate numerically the power expression at each ↑ node, using values in the VAL array. For example, simplifying the tree shown in Exercise 6 would yield the following tree expression:



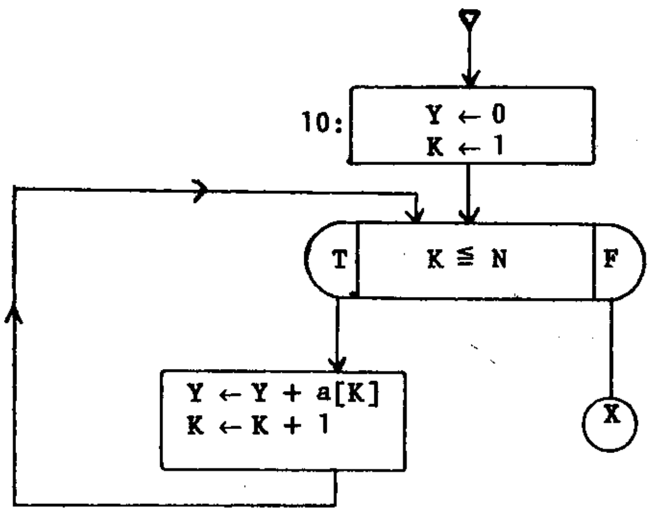
3. PROCEDURES AND SUBROUTINES

A. Subroutine Linkage

You may recall from Section 2 on flow charts that an entire flow chart may be used as a single computation box in another "higher level" flow chart; conversely, any computation box may contain a (non-primitive) sub-process which is defined by an entire lower level sub-flow chart. Thus, a complicated computing process can be represented at all levels of detail by a whole series of flow charts, sub-flow charts, etc. As an example, suppose we have defined the summation process:

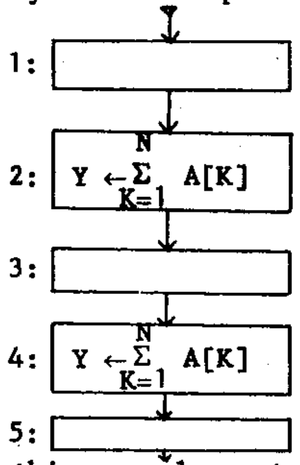
$$Y \leftarrow \sum_{K=1}^{N} A[K]$$

by the flow chart:



(A1)

This identical process, although defined only once, can be "used" (i.e., executed) in any number of places in a higher level flow chart, as in the example:


1:
2:
$Y \leftarrow \sum_{K=1}^{N} A[K]$
3:
4:
$Y \leftarrow \sum_{K=1}^{N} A[K]$
5:

(A2)

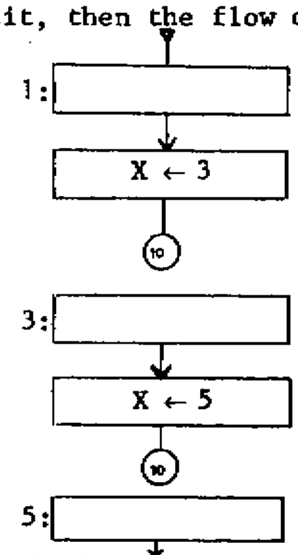
Boxes 1, 3, and 5 in this example contain other computing processes, which could be themselves entire complex flow charts; since the particular processes are irrelevant, we have simply left these boxes empty.

This notation allows the same subprocess to be executed at several points in the overall flow chart (A2) while the detailed flow chart (A1) which actually defines the subprocess need be drawn only once. Thus, the subprocess notation can save writing and simplify the form of a flow chart.

When we discussed primitive flow charts, we gave explicit (and simple) rules for their execution. The basic idea was contained in the "fetch/execute" cycle, which required the determination of a successor to each box executed. This successor is quite obvious in a primitive flowchart: one merely follows the unique arrow leaving each box. However, if we examine the explicit successor rules implied by the example flowchart (A2) above, we find that a new kind of successor relation has been introduced. Both box 1 and box 3, which precede executions of the sub-flowchart (A1), clearly have the sub-flow chart itself (that is, the first box of the sub-flow chart) as their successors; however, the sub-flow chart does not have a fixed successor. We can think of X in the exit symbol as a variable whose value is the (numerical) label of the successor box. Executing the sub-process as a single box of the main flowchart involves two steps:

(1) Setting the "exit variable" X equal to the label of the next box in order in the main flowchart (A2);

(2) Executing a go to box, so that the first box of the sub-flow chart (A1) will be executed next. When execution of the sub-flowchart has finished, the go to box " (X) " returns to execute the next box in (A2) after the place which jumped off to the sub-flowchart. Thus, if the successor relation is made explicit, then the flow chart (A2) would become:



A part of a program which corresponds to the sub-flowchart (A1) is called a subroutine. In general, a subroutine is:

> a segment of program which, once defined, can be executed as a single "statement" or instruction from any place in a "master" program.

The master program may in turn be a subroutine for an even higher level program, etc; subroutines, therefore, introduce into programming a hierarchy of levels of command corresponding to the levels of detail of flowcharts. No matter how complex its definition, a subroutine is executed as a single statement; thus, the subroutine definition effectively creates a new elementary or "primitive" process for the master program.

Like the sub-flow chart discussed earlier, the subroutine has a variable successor which must be set each time the subroutine is executed. A subroutine is executed, or "called", by the main program using the same two-step process used to execute a sub-flowchart:

(1) Set the exit "label" (actually, the next memory address of the main program);

(2) Execute a "go to" which transfers control to the first instruction of the subroutine.

This two-step process is frequently called a 'subroutine jump". At the completion of execution of each subroutine a "variable <u>go</u> <u>to</u>" is executed, returning control to the first instruction in the main program past the point from which the subroutine jump was performed.

The programming mechanism which realizes the subroutine jump and the subsequent return to the next instruction in the master program is called "subroutine linkage". The subroutine linkage is so important that most computers include in their set of primitive operations a "subroutine jump" command which sets the "exit label" (the address of the next instruction in memory) and also transfers control to the subroutine.

Notice that the master routine and the subroutine need only agree upon the "name" (that is, memory address) of the variable X which holds the exit label, and upon the first location of the subroutine itself. Except for these agreements, the subroutine is independent of its master program.

We can now cite two important advantages of subroutines for programming:

(1) Subroutines save repetitive writing of identical program segments.

(2) Subroutines provide a natural way of dividing a complex program into sub-tasks, both parallel and nested, which can be programmed and debugged independently of the other sub-tasks.

B. Procedures: ALGOL Subroutines

Every important computer programming language has provision for subroutines. In ALGOL, subroutines are called <u>procedures</u>, to emphasize their performance of independent tasks. The structure of ALGOL procedures is very general and powerful, and is one of the most significant features of the ALGOL language. We will now concentrate upon ALGOL procedures; bear in mind, however, that most of the principles of procedures are applicable to the subroutines of other programming languages.

Each procedure has a name, which has the form of an identifier. Like most other kinds of ALGOL identifiers, a procedure identifier must appear in a declaration -- a procedure declaration -- at the beginning of the program. A procedure declaration does more than indicate that the name is a procedure identifier, however: the declaration also gives the definition of the procedure. Thus, the declaration contains a "body" of one or more statements which will be executed when the procedure is called. It is important to understand that the statements within the body of the procedure declaration are executed only when the procedure is called, not when it is declared. Corresponding to each procedure declared in a program, the ALGOL compiler assigns a memory cell to contain the variable exit address X. Each call for the procedure is translated into machine language as a subroutine jump instruction.

There are two distinct ways in which an ALGOL procedure can be used:

(1) To execute a complete computational process - i.e., as a single flow chart box; or

(2) To define a value - i.e., to be used as an operand in an arithmetic expression.

A procedure which performs a complete process is executed as a statement, called a "procedure call statement". The execution of a procedure call statement means simply a subroutine jump to the procedure declaration. Following execution of the body of the declaration, control returns to execute the next statement after the call statement.

For example, here is part of an ALGOL program using a procedure DOSUM which performs the algorithm of flow chart (A1):

```
              begin
                   real Y; integer K, P;
                   real array  A[1:100] ;
heading}           procedure  DOSUM ;
                       begin
                            Y ← 0 ;
procedure                   for  K ← 1  step  1  until  N  do
declaration                      Y ← Y + A[K] ;
                       end DOSUM ;
```

```
procedure  ⎧          ⋮
call       ⎨       DOSUM ;
statements ⎪          ⋮
           ⎪       if P > 3 then  DOSUM  else  N ← 0 ;
           ⎩          ⋮

     end  Procedure Example 1;
```

Here the procedure declaration has the form:

$<$ procedure declaration $> \sim$ $<$ heading $>$ $<$ body $>$.

The $<$ body $>$ can be any simple statement or compound statement (or, more accurately, any block). The simplest form of $<$ heading $>$ is:

procedure $<$ procedure identifier $>$;

The final end of the procedure body (or the semicolon after the entire declaration if there is no final end) implies a subroutine return to the first statement in the master program after the procedure call statement, which consists simply of the procedure identifier DOSUM.

A procedure which defines a value is called a function, and its name is a function designator. To make DOSUM into a function designator, we place the type declarator word real before the word procedure in the declaration. Somewhere in the body of the procedure, we assign the desired result to the procedure name DOSUM.

```
     real  TEMP ;
     real procedure  DOSUM ;
          begin
               TEMP ← 0 ;
               for  K ← 1  step  1  until  N  do
                    TEMP ← TEMP + A[K] ;
               DOSUM ← TEMP
          end  DOSUM ;
```

This declaration defines a function designator DOSUM which can be used as an operand in any arithmetic expression. Evaluating the operand means a call for the procedure; when control subsequently passes the final end of the procedure body, it returns to continue evaluation of the arithmetic expression, with the value which has been assigned to DOSUM being taken as the value of the operand. For example, execution of the statement:

```
Y ← 3 * DOSUM ↑ 2 + COS(DOSUM) ;
```

will cause two separate executions of DOSUM; since the same sum will be computed twice, this is an inefficient use of function designators.

We have discussed the idea of a subroutine and the simplest kind of procedures. There are several other important concepts in ALGOL procedures which we will now mention but not discuss fully. McCracken's text introduces these ideas on a fairly simple level; for more complete treatment of procedures, we recommend the tutorial article on ALGOL by Bottenbruch and the book by Dijkstra listed in the Bibliography, Part V.

The power and usefulness of ALGOL procedures lies in two abilities they give to a programmer:

(1) An algorithm can be generalized by parametrization. Thus, an algorithm can be developed to perform any one of an entire family of related computing tasks; the particular task is selected by the choice of <u>actual parameters</u> in the procedure call, which are substituted for the dummy names or <u>formal parameters</u> in the procedure declaration.

(2) A procedure can be a "black box", an independent program to which the programmer can communicate only via parameters. Once declared, it can be called with appropriate inputs via actual parameters and results returned via parameters and perhaps a function designator, without otherwise disturbing the calling program. Furthermore, the programmer who writes the call for such a procedure might not even understand the algorithm used by the declaration, yet he can use the algorithm if he knows what actual parameters are required in the call. In effect, he passes inputs through slots in the top of the box and turns the crank on the side; after a certain amount of wheezing and groaning, the black box returns the results through another slot.

One important use of "black box" procedures is for algorithm publication; all algorithms which are published in ALGOL are in fact written as "black box" procedures. The published procedures can be kept in libraries of useful algorithms; their declarations can be inserted into any program (either symbolically or in a suitable machine language form obtained from previous compilation) and called as needed.

Making a procedure a black box requires three concepts:

(1) The subroutine, already discussed.

(2) Parametrization.

(3) Declaration of ALGOL names which are purely <u>local</u> to the procedure, as opposed to <u>global</u> variables such as K and N in DOSUM. Notice that execution of DOSUM changes K in the master program. If the programmer has also used K for another purpose, this change of K by DOSUM could be very annoying! By declaring K to be local to the procedure, he can create two independent K's and avoid an undesirable conflict.

C. Exercises on Procedures.

<u>Exercise 1</u>: (Specifiers and Name vs. Value)

We wish to generalize from the following <u>for</u> statement:

$$\begin{aligned}&\text{L1: } \underline{\text{for}}\ \boxed{\text{I}} \leftarrow \boxed{\text{I+1}}\ \underline{\text{step}}\ \boxed{\text{3 * I}}\ \underline{\text{while}}\ \boxed{\text{I} < 73}\ \underline{\text{do}}\\ &\qquad\qquad \boxed{\text{A}}\ [\boxed{\text{I}}] \leftarrow \text{sin}\ ((\boxed{\text{I/100}}))\ ;\end{aligned}$$

by letting all 8 of the circled syntactic units be formal parameters and making it a procedure FORSTATE. With suitable calls, this procedure will perform either the statement labeled L1 above, or the statement L2 below.

$$\begin{aligned}&\text{L2: } \underline{\text{for}}\ \text{X} \leftarrow \text{A[I]}\ \underline{\text{step}}\ -.01\ \underline{\text{while}}\ \text{X} > 0 \wedge \text{sin (X)}\\ &\qquad\qquad < \text{X/2}\ \underline{\text{do}}\ \text{A[A[I]]} \leftarrow \text{sqrt (abs(1 - X))};\end{aligned}$$

Assume that the variables in L1 and L2 have been declared by:

<u>real</u> I, X ;

<u>real</u> <u>array</u> A[1:100] ;

Note that <u>sin</u> and <u>sqrt</u> are function designators: i.e., real procedures.

Copy the following declaration of FORSTATE and fill in the specifiers for all formal parameters, and the value part (if any). Make as many parameters called-by-value as possible without losing the ability to handle the L1 and L2 statements.

```
procedure  FORSTATE (A, B, C, D, E, F, G, H) ;
     begin
          for  A ← B  step  C  while  D  do  E [F] ← G(H) ;
     end  FORSTATE ;
```

Exercise 2: (Name vs. Value Parameters)

An ALGOL program begins as follows; the statement labeled L3 has been left blank.

```
     begin
          real  I, J ;
          real array  A[1:100] ;
          real procedure  PHI(I) ; real I ;
               begin
                    I ← I+1 ; PHI ← A[I+2]
               end  PHI ;
          real procedure  FOE (I) ; value I ; real I;
               begin
                    I ← I+1 ; FOE ← A [I+2]
               end  FOE;
L1:  for  J ← 1  step  1  until  100  do  A[J] ← 2*J ;
L2:  I ← 3 ;     L3: [          ]
```

Suppose one of the following six statements is chosen as statement L3; give the values of the variables I and J in the main program after execution of L3.

```
     (a)  L3:  J ← PHI (I)
     (b)  L3:  J ← PHI (A[I])
     (c)  L3:  J ← PHI (I) + PHI (I)
     (d)  L3:  J ← FOE (I)
     (e)  L3:  J ← PHI (A [I])
     (f)  L3:  J ← FOE (A [I])
```

Exercise 3:

The procedure WOE has been declared as follows:

```
          real procedure  WOE (B, GONE) ;
               real  GONE, B;
               WOE ← if  B < GONE  then  B↑2  else  B - GONE ;
```

Assume X = 3 and Y = 2. Find the value of the following arithmetic expression:

```
     WOE(X, 2*Y) ↑ 2 / WOE (if  X = Y  then  X  else  2 * Y, X - Y).
```

Exercise 4:

Write a procedure named MAX, which, given a set of n points, finds the distance between those two of the points which are farthest apart. MAX is to be a function designator whose value will be the desired distance. It is to have three formal parameters, as follows:

X: an array with subscripts from 1 to N,

Y: an array with subscripts from 1 to N,

N: an integer - the number of points.

The coordinates of the kth point are (X[k], Y[k]).

Exercise 5:

The following ALGOL procedure is declared in an ALGOL program. In the calls for TABLE(A,X,Y), the actual parameter substituted for A will be a complicated expression.

Rewrite the procedure declaration so that its execution will require as little time as possible, while leaving the same values in the input and output parameter areas. You may declare additional local variable storage.

```
procedure  TABLE (A,X,Y) ;
real  A;  array X,Y ;
     begin  integer I ;
            integer array  Z[1:40] ;
            for I ← 1  step  1  until  40  do
                 Z[I] ← I ;
            for I ← A - 1  step  1  until  2*A + 9  do
                 Y[I+1] ← -Z[I+1] ↑ 3/(X[I+1] ↑ 2 + A↑2)
                 ↑ (1/2) - Z[I+1] ↑ 6/(3*(X[I+1] ↑ 2+A ↑ 2)
                 ↑ (3/2)) + SQRT(A↑2 + 1) + 3.2 * 10 ↑ -2
     end ;
```

4. BACKUS NORMAL FORM: LANGUAGE vs. META-LANGUAGE

A reference manual for a programming language should define the language precisely, explicitly, and unambiguously. It must specify both:

(1) the rules of grammar, or syntax, for legal programs in the language, and

(2) the meaning, or semantics, of any legal program.

In the past, programming manuals have fulfilled these objectives by giving descriptions, in English, of the language and its meaning. As computer languages have become more complex, however, such descriptions have become less and less adequate. Even well written descriptions have frequently been found, on close examination, to leave ambiguous certain points about the language being defined.

From an abstract viewpoint, the definition of the syntax is simply a set of rules by which one can decide whether any given string of characters is, or is not, a legal program in the language. The definition of the syntax must first specify (by listing it) the alphabet of the language. It must then show how the larger elements of the language are made up from the alphabet. It is here that a difficulty arises. The alphabet of the language being defined (the "object language"), overlaps the alphabet of the language in which the definition is written (the "meta-language"); thus the reader frequently cannot tell whether a given piece of text is an example of the object language or is a part of the descriptive meta-language.

With these considerations (and others) in mind, J. W. Backus[1,2] has devised a technique for specifying the syntax of programming languages. He has defined a language, which has come to be called "Backus Normal Form" (abbreviated "BNF") to be used as a meta-language for the clear and unambiguous definition of object languages. Indeed, BNF was devised specifically for the purpose of describing ALGOL-60 syntax. The following paragraphs contain a description of BNF with some examples of its use.

1 J. W. Backus, The Syntax and Semantics of the Proposed International Language of the Zurich ACM-GAMM Conference. ICIP Paris, June 1959.

2 J. W. Backus, et al, Revised Report on the Algorithmic Language ALGOL-60. Communications of the Association for Computing Machinery, Vol. 6, No. 1, (January 1963), 1-17.

In BNF the following four meta-linguistic symbols are introduced:

< > | ::=

These are called 'meta-linguistic symbols' since they must not be included in the alphabet of the object language. The characters < and > are used as brackets to surrounding strings of meta-language characters. Such a bracketed string of characters is called a meta-linguistic variable and is the name of a class of strings in the object language.

The mark | may be read as "or", and the mark ::= (to be regarded as a single symbol) may be read as "is defined as". The use of these symbols is best illustrated by example.

Example 1:

```
< digit >  ::=  0|1|2|3|4|5|6|7|8|9
```

This meta-linguistic formula may be read:

"A member of the meta-linguistic class < digit > is defined as 0 or 1 or 2 or 3 or 4 or 5 or 6 or 7 or 8 or 9." Thus an occurrence of < digit > in a meta-linguistic formula stands for any one decimal digit.

An especially useful feature of BNF - indeed, its reason for existence - is that a definition using this meta-language may be recursive. A recursive definition, for our purposes, may be thought of as one in which the object being defined is included, either directly or indirectly, in its definition.

Example 2:

```
< integer >  ::=  < digit > | < integer >  < digit >
```

The meta-linguistic formula in this example may be read as:

"A member of the meta-linguistic class < integer > is defined as (either) a member of the meta-linguistic class < digit > , or a member of the meta-linguistic class < integer > followed by a member of the meta-linguistic class < digit > ."

More briefly, one might say:

"An < integer > is either a < digit > or an < integer > followed by a < digit > ." The recursive nature of this definition of < integer > is seen in the occurrence of the meta-linguistic variable < integer > on both the left side and the right side of the formula. Let us now consider a more complex case.

Example 3:

< ab > ::= (|[| < ab > (| < ab > < digit >

This meta-linguistic formula defines the class < ab >. The reader should satisfy himself that the following are members of < ab >

[(((1(37(

(12345(

(((

[86

and that the following are not members of < ab >

213

[[[

[[

The following example, taken from the ALGOL-60 Report [2], illustrates the use of recursiveness for the definition of the syntax of < arithmetic expression > . Notice how the definitions are built up in steps corresponding to the rules of precedence of the operators: a < primary > is evaluated before a < factor > which is evaluated before a < term > which is evaluated before a < simple arithmetic expression > , etc. Thus, the syntax has been defined to parallel the semantics of the language.

Example 4:

< adding operator > ::= + | -

< multiplying operator > ::= × | / | ÷

< primary > ::= < unsigned number > | < variable > |
< function designator > |
(< arithmetic expression >)

< factor > ::= < primary > | < factor > ↑ < primary >

< term > ::= < factor > | < term > < multiplying operator >
< factor >

< simple arithmetic expression > ::= < term > |
< adding operator > < term > | < simple
arithmetic expression > < adding operator >
< term >

< if clause > ::= if < Boolean expression > then

< arithmetic expression > ::= < simple arithmetic
expression > | < if clause > < simple arithmetic
expression > else < arithmetic expression >

A careful examination of this syntax will show that the following are correct instances of the indicated meta-linguistic classes:

< primary >

```
        6.02₁₀23
        THETA
        cos (THETA ↑ 2)
        (3*Y + THETA ↑ 2)
```

< factor >

```
        THETA
        THETA ↑ 2.3
        THETA ↑ Y ↑ cos(THETA ↑ 2)
```

< term >

```
        Y
        3*Y
        THETA * Y / (3*Y + THETA ↑ 2)
```

< simple arithmetic expression >

```
        3*Y
        3*Y + THETA ↑ 2
        THETA ↑ 2/3.2 ↑ Y + cos(THETA ↑ 2) - 3*Y
```

< arithmetic expression >

```
        3*Y + THETA ↑ 2
        if Y > 3  then  3*Y  else  THETA ↑ 2
        if Y < 3  then
                  (if THETA >0 then  3*Y else  Y + 2)
                  else  THETA
```

__Exercise 1:__

ALGOL programs are constructed of the following syntactic units:

1. Array Declaration	7. Number
2. Type Declaration	8. Simple Variable
3. If Statement	9. Subscripted Variable
4. For Statement	10. Arithmetic Expression
5. Assignment Statement	11. Boolean Expression
6. Go To Statement	12. Conditional Arithmetic Expression

For each of the particular ALGOL constructions listed below, give the number(s) of <u>all</u> the syntactic units listed above, of which the <u>entire</u> construction (not its constituents) is an example.

a. X1

b. A[P, Q]

c. A[P, Q] ← SIN (P + Q↑2) ↑ 2

d. <u>if</u> P > Q <u>then</u> P + 2 <u>else</u> Q + 2

e. P = Q

f. <u>real</u> P, Q, X1

g. <u>if</u> P > Q <u>then</u> X1 ← P + 2 <u>else</u> X1 ← Q + 2

h. <u>real</u> <u>array</u> A[1:10, -5:5]

i. X1 ← (<u>if</u> P > Q <u>then</u> P <u>else</u> Q) + 2

j. X1 > (<u>if</u> P > Q <u>then</u> P <u>else</u> Q) + 2

k. GOTOL ← GOTOL + 3

<u>Exercise 2</u>:

(a) Consider the following meta-linguistic classes:

< pop > ::= (< pop >, < primary >) | < primary >

< primary > ::= < letter > | [< pop >]

< letter > ::= a|b|c|d|e|f|g|h|i|j|k|l

For each of the following, indicate whether or not it is a < pop > and whether or not it is a < primary > .

a	(c,d)	([a,b])
[a]	((a,b), c)	[[[a]]]
(a)	(a, (b, c))	((a, b), (c, d))
[a,b]	[(a, b])	((a, b), [(c, d)])

(b) Consider the following meta-linguistic classes in addition to those defined in (a):

```
< cat > ::= (|) | , | < cat > < cat >

< dog > ::= [|] | < empty >

<fish>  ::= <cat> | <dog> | <cat> <dog> | <dog> <cat> |
            <cat>   <dog>   <cat>

< bop > ::= < fish > a < fish > b < fish > c < fish >
```

It is clear that there are < bop >s which are not < pop >s and < pop >s which are not < bop >s. List all < bop >s which are < pop >s.

(c) Consider the following:

```
< cow > ::= (|) | [|] |, | < cow > < cow >

< boop > ::= < cow > a < cow > b < cow > c < cow >
```

< Boop >s include more than < bop >s. Write a < boop > which is not a < bop >. Are there any < bop >s which are not < boop >s? Why would it have been undesirable had you been asked to list all < boop >s which are < pop >s? Give an example. (This can easily be answered in one sentence.)

PART IV - PROGRAMMING PROBLEMS AND THE TEACH SYSTEM

1. THE USE OF TEACH PROCEDURES FOR STUDENT PROGRAMMING PROBLEMS

For each programming problem assigned in the course, the instructor can write a corresponding "TEACH" procedure to be placed into the ALGOL program library.* When called from the library by a student's program, a TEACH procedure supplies several different sets of input data to the program and checks whether the results it computes are correct. TEACH also prints the data supplied, the student's results, the correct answers if the student's are wrong, perhaps some diagnostic information to aid the student in finding his error(s), and a score for the problem.

The TEACH procedures are used both by the students to help debug their programs and by the instructors to grade the finished product. Thus, a student will normally prepare his program (in the form shown below) to call the TEACH procedure; during each debugging run, TEACH will supply test data and check his answers. When he is satisfied that the program is working correctly, the student hands it in, still set up to call TEACH. All the program decks for the problem are then batched and run for grading by the instructors, and the scores printed by TEACH during this run are recorded as the grades for the problem.

During the grading run, the TEACH procedure is modified to supply different data values ("grading data") than the values ("debug data") supplied to the student for his debugging runs. The grading data, of course, is chosen to thoroughly test the program and includes all special and boundary cases.

For the first few problems, the debugging data generally gives a fairly complete test of the program; for the later ones, however, the debugging data is sufficient only to ensure that the solution generally does what is desired. This is to encourage the students to complete the debugging process themselves, supplying their own data sets (in a manner explained below) to test all special cases. When the deck is handed in for grading, however, it must contain none of the student's own test statements but must be set up in the standard manner shown below to call the TEACH procedure for grading.

* This section describes the TEACH procedures as they are used in the S205 course at Carnegie Institute of Technology.

The following example shows a student's deck set up to call the procedure TEACH4. The statement of a TEACH problem details the formal parameters to the corresponding TEACH; the student can choose the actual parameter names as he wishes (although most students simply use the formal parameter names).

```
Card No.        Column 1, 2        Column 4...72

   1                               comment  JOHN DOE, < student number >,
                                            PROBLEM 4;
   2                               begin
   3                                   real  X, XMIN; \
                                                       } Student's Declarations
   4                                   integer I ;    /
   5               SY                  LIBRARY TEACH4        "System Card"
   6                               GRADE:  TEACH (< student number >, X, XMIN);
                                                           "Call" TEACH4
   .
   .                                  (Now comes his program, \  Student's
   .                                   followed by:)          /  Program
  n-1                                  go to GRADE ;
   n                               end
```

When this program is executed, the following sequence of events occurs:

(1) TEACH4 is called by the statement on card 6.

(2) TEACH4 executes assignment statements which place a set of input data values into the proper input variables.

(3) TEACH4 returns control to the student's program which then executes with the data in the input variables.

(4) "go to GRADE" returns control to TEACH4 which checks the results which the student's program has left in the output variables. The input data and the student's answers are printed along with a message telling which, if any, results are wrong. TEACH4 also prints its own answers if they differ from the student's answers.

(5) Steps (2), (3), and (4) are repeated for each data set. Finally, TEACH4 concludes with the message:

```
STUDENT NUMBER < student number >  SCORES < score >
                OUT OF < number >
```

If the student's program creates a run-time error - e.g., an "exponent overflow" (which includes dividing by zero) or an illegal argument to a transcendental subroutine, control will return to the TEACH procedure. The current data set will be marked wrong, a suitable message will be printed, and the next data set will be supplied.

Each TEACH procedure should allow the student's program to compute for a fixed maximum time on each data set. If control has not returned to TEACH at the expiration of that time, the current data set will be marked wrong, a message will be given, and the next data set will be supplied.

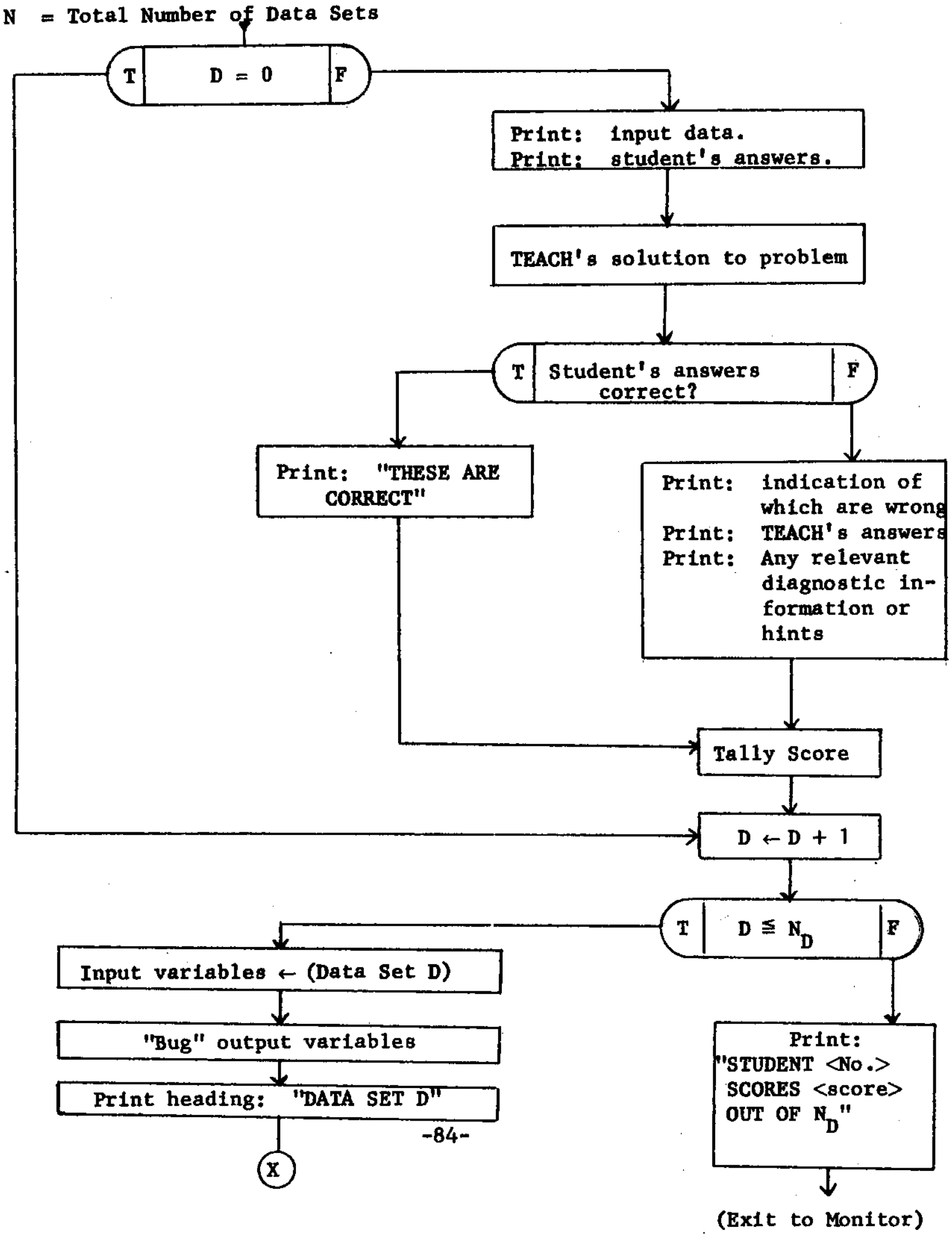
In the example program shown above, card 5, with "SY" in columns 1 and 2, is a "SYstem" card which causes the TEACH4 procedure to be loaded into the G-20 along with the student's program. The details of the manner in which this is done are not relevant to the present discussion.

The standard structure of a TEACH procedure is shown in more detail in the flow chart which follows.

Simplified Flow Chart for TEACH Procedure

D = Data Set Number (initially = 0)

N = Total Number of Data Sets


T
D = 0
F
Print: input data.
Print: student's answers.
TEACH's solution to problem
T
Student's answers correct?
F
Print: "THESE ARE CORRECT"
Print: indication of which are wrong
Print: TEACH's answers
Print: Any relevant diagnostic information or hints
Tally Score
D ← D + 1
T
D ≦ N_D
F
Input variables ← (Data Set D)
"Bug" output variables
Print heading: "DATA SET D"
X
Print: "STUDENT <No.> SCORES <score> OUT OF N_D"
(Exit to Monitor)

There are several aspects of this flow chart which are worth noting.

1. The variables D (= Data Set number) and SCORE (the cumulative score for the run) should be _own_ variables, local to TEACH*.

2. In most cases, the TEACH procedure contains its own (presumably) correct solution to the problem; this solution is executed each time TEACH is re-entered, to determine the correct answer. We could, of course, pre-compute the answers and store them in a file on disc or tape memory from which they could be retrieved for each student run. However, the actual computing time for solving a TEACH problem is usually negligible; most of the time is spent in compiling and printing. Therefore, the simpler technique of re-computing the correct answers each time is generally quite efficient.

   Furthermore, re-computing the answers each time provides considerable flexibility in posing problems. For example, some TEACH procedures written at Carnegie Institute of Technology have contained non-constructive solution-_checking_ algorithms rather than solution algorithms paralleling the student's program, either because the problem did not have a unique answer or because checking was far simpler than computing the answers. Forcing a problem to have a unique answer sometimes requires extraneous assumptions and restrictions to be explained to (and comprehended by) the student; in such cases it is better to simply let the result be non-unique.

3. The correct solution (or solution-checker) within TEACH takes its input values from the same variables (or, more accurately, from the formal parameters corresponding to the actual parameters) which are used by the student's program, although TEACH must use a different (and local) set of output variables. The correct solution (or the solution-checker) is executed _after_ the student's program has executed; therefore, the student may supply his own data for debugging via statements assigning new values to the input variables, after calling TEACH; TEACH will then use the student's data to check his answers, and will print his data and results. A result of this provision is that the student's program must not change the input variables in the process of solving the problem.

   Of course, it is not necessary for the student to use TEACH at all for debugging runs; however, if he does not use TEACH, he must write all his own print statements. At least for the first few problems, printing the data and results requires more complicated programming than does solving the problem itself; hence, the student is well-advised to let TEACH do the printing.

---

* It is interesting to note that our TEACH procedures are able to use these _own_ variables only because ALGOL-20 initializes all _own_ variables to zero.

4. Before it returns control to the student's program, TEACH normally "bugs" the output variables with distinctive values so that when next re-entered it can test for the student's failing to initialize variables, assigning no value to an output variable, or storing into elements of any output array which are not included in the limits of the data. In each case, appropriate diagnostic messages can be printed.

It should be apparent that the TEACH procedures provide only a very simple measure of the quality of a student's program: whether or not it works correctly. Unfortunately, it cannot judge the clarity, elegance, simplicity, cleverness, or sophistication of a program. On the other hand, it is possible to choose data sets and design a TEACH procedure to judge a program more finely than simply: "right or wrong".

For example, for most problems there exist (and some students will discover) woefully inefficient algorithms; these can be detected by a suitable choice of the time limits for the data sets. By setting the time limit at two or three times the time required by TEACH's solution, we can detect poor algorithms as distinguished from minor differences of inefficiency due to trivial coding variations. We feel that this emphasis upon generally efficient algorithms rather than upon coding details is appropriate for a college programming course. Furthermore, in actual programming factors of two in efficiency are usually important only in inner-most loops; more frequently factors of 10, or of N/log N are the important considerations in programming.

Choosing data sets for grading a problem requires a great deal of care. Generally one gives a number of data sets, each of which tests some different aspect, special case, or boundary condition of the problem. It is important to keep these sets as nearly "orthogonal" i.e., independent, as possible, so that the total score will be a true and fair reflection of how thoroughly the student understood and solved the problem. If data sets are chosen badly, all students will get either 100 or 0 on the problem; if they are well-chosen, on the other hand, the scores will show a normal distribution - although in practice it is very difficult to avoid a skew towards the high end, since it is hard to choose a problem which most students will be able to do partially but which only a few students can do perfectly.

2. A COLLECTION OF PROGRAMMING PROBLEMS

This section contains a collection of programming problems - or more accurately, kernels of programming problems - suitable for an introductory programming course at the college level.* Most of the pro-

* Many of the problems stated below have, in fact, been given to students in the course S205 at Carnegie Institute of Technology over the past four years.

blem statements contained here would (or did) need significant expansion or elaboration before they could be assigned to a class. The problem as stated to the students ought to include a careful definition of unfamiliar terminology, perhaps illustrative diagrams and drawings, a complete specification of the parameters to be used to call the TEACH procedure or equivalent control program, and explicit statements concerning possible ambiguities in the problem statement.

Furthermore, an instructor who uses one of these "problem kernels" as a basis of a programming problem is likely to want to increase or decrease the difficulty of the problem to match his students, the level and purpose of the particular programming course, and his own feelings about proper pedagogical technique. The perceptive reader should have no trouble thinking of variations, perhaps endless variations, on each of the problems listed here.

The list has been categorized (approximately) into the following four groups:

A. Combinatorial problems,

B. Geometry problems,

C. Representation problems,

D. Numerical Computation problems.

It will be clear that many of them could appear in several of these categories, but this division still gives some useful indication of the nature of the problems.

A. Combinatorial Problems.

A1. "The Change Problem":

Suppose you have an unlimited number of coins of denominations 1, 2, 5, 10, 25, and 50 cents. Given a number of cents C, compute how many <u>different ways</u> there are to total C cents using only coins of these denominations.

How is the algorithm altered if the coins have a general set of denominations $\mu_1, \mu_2, \ldots, \mu_k$, with $0 < \mu_1 < \mu_2 < \ldots < \mu_k$?

A2. Compute all partitions of a given integer number n; i.e., find all sets of positive integers which sum to n.

A3. Count the unique numbers in a list of n numbers.

A4. Suppose you have N objects labeled 1, 2, ..., N, placed counter-clockwise in a circle. Starting at object labeled K, tag the Lth object around the circle; tag the next Lth remaining object, and continue tagging every Lth object (of those remaining untagged) until only one remains. If an object is already tagged, shift counter-clockwise until the first untagged one is encountered or there are none left untagged. Write a program which, given values for N, L, and K, determines the label of the last object tagged.

A5. You are given a sequence of M numbers $X_1, X_2, \ldots, X_M$; from these M, your program should select a subsequence of N numbers: $X_{i_1}, X_{i_2}, \ldots, X_{i_N}$. The subsequence must satisfy all of the following conditions:

(a) Adjacent numbers have opposite signs:

$$X_{i_k} * X_{i_{k+1}} < 0 \text{ for } k = 1, 2, \ldots, N-1.$$

(b) The order of the original set is preserved:

$$i_p < i_q \text{ if and only if } p < q.$$

(c) The difference between largest and smallest members:

$$\max_k (X_{i_k}) - \min_k (X_{i_k})$$

must be as large as possible.

Furthermore, the value you use for N must be the largest integer for which properties (a) and (b) and (c) hold. Given M and $X_1, \ldots, X_M$, your program should compute N and $i_1, \ldots, i_N$.

A6. "The Calendar Problem":

Write a program to compute how often the Fourth of July falls on a Thursday. Assume a calendar with the following properties: 365 days per year with 7 days per week; starting at year 1:

a. Every Lth year is a leap year (i.e., it has 366 days, the extra day being February 29) except -

b. Every (L * M)th year is an ordinary year (i.e., not a leap year) except -

c. Every (L * M * P)th year is a leap year.

This calendar is simply a generalization of our Gregorian calendar which uses L = 4, M = 25, P = 4.

Suppose we number the days of the week as follows:

Sun	Mon	Tue	Wed	Thu	Fri	Sat
0	1	2	3	4	5	6

With this information, we may state the exact problem.

For the span of years 1, 2, ..., (L*M*P) and assuming the Fourth of July falls on day D in year 1, compute how many times the Fourth of July falls on a Thursday.

Notice that in the special case where L, M, and P have the values for our Gregorian calendar, the following are true:

(a) The product (L*M*P) is 400. The answer to the problem is not simply 400/7 because, e.g., this is not an integer. In other words, the Fourth of July does not fall with equal frequency of each day of the week.

(b) Over the entire 400 years, there are the same number of each day of the week. This means the Gregorian calendar is cyclic, i.e., repeats itself every 400 years (but not sooner!). Thus, the same answer would result independent of the year with which we started.

A7. Program one of the algorithms for ordering or "sorting" N numbers. The common elementary algorithms include:

(a) Exchanging, also called "shuttle sort": Exchange members of each adjacent pair which is out of order, repeating until entire set is sorted.

(b) Linear selection with exchange: This is the algorithm used in the lecture notes in Section 1 of Chapter II.

(c) Radix (or "pocket") sort: Sorting digit-by-digit in the manner of an IBM card sorting machine.

(d) Binary merge sort: forming sorted substrings of length 2, 4, 8, 16, ..., 2↑ entier (LN(N)).

In addition, there are non-elementary sorting techniques involving tree structures which form a plausible introduction to list processing.

References on sorting techniques:

Sorting on Electronic Computers, E. H. Friend, JACM, 3, pp. 134-168 (July 1956).

Analysis of Internal Computer Sorting, Ivan Flores, JACM, 8, pp. 41-80, (January 1961).

Trees, Forests, and Rearranging, P. F. Windley, Computer J., 3, pp. 84-98, (July 1960).

A8. This is a sequence of four simple problems which lead the student naturally to writing a recursive procedure in part (d). If recursion is not available, the statement of part (d) can easily be modified to instruct the student to do his own stacking of parameters and an exit switch. A solution to part (d) is given.

It may be of interest that this problem is related to a famous unsolved question in number theory which is being empirically tackled on computing machines. See, for example: Fraser, W., and Gottlieb, C. C., A Calculation of the Number of Lattice Points in the Circle and Sphere, Mathematics of Computation, 16 (1962), pp. 282-290.

(a) A circle of radius r is centered at the origin of a plane Cartesian coordinate system. Each point (u, v) in this plane whose coordinates u and v are positive, zero, or negative integers is called a "lattice point."

Write an ALGOL procedure CIRCL to compute L, the number of lattice points lying within or on a circle of radius r centered at the origin. That is, count the number of pairs (p, q) of integers such that $p^2 + q^2 \leqq r^2$. Make CIRCL a function whose value is L with the value of r^2 (not r) as a formal parameter.

Count points only for one quadrant, but be careful not to count the axes or the origin point more than once.

(b) A sphere of radius r is centered at the origin of a three-dimensional Cartesian coordinate system. A "lattice point" in this space is any point (u, v, w) whose coordinates u, v, and w are integers.

Write an ALGOL procedure SPHER to compute M, the number of lattice points in three-dimensions which lie within or on the surface of a sphere of radius r centered at the origin.

Make SPHER a function having r^2 as a formal parameter. Use CIRCL as a subroutine in SPHER. Count the points in the northern hemisphere only, doubling the result to include the "southern hemisphere"; but be careful not to count the points in the equatorial plane twice.

(c) A four-dimensional hypersphere of radius r is centered at the origin of a four-dimensional coordinate system. Every quadruplet (u, v, w, x) of integers defines the coordinates of a lattice point in this four-dimensional space.

Write an ALGOL procedure HYPER to compute N, the number of lattice points lying within or on the surface of the four dimensional hypersphere of radius r. Make HYPER a function with the value of r^2 as a formal parameter.

Use SPHER as a subroutine in HYPER. Count the points in the "northern hemihypersphere" but be careful not to count the "equatorial" sphere twice.

(d) Write a recursive ALGOL procedure POINT (P, n). POINT counts the lattice points in n dimensions which are within or on the surface of an n-dimensional sphere of radius $r = \sqrt{P}$. For example, POINT (P, 2) should give the same result as CIRCL (P). POINT should be a recursive procedure calling itself to count points in n-1 dimensions, etc., just as HYPER used SPHER which in turn used CIRCL.

Make POINT a function. Count points in only half of each space, subspace, sub subspace, ..., and double; but don't count the middle points twice.

Note: Here is a solution to Part (d):

```
    integer procedure  POINT (RHO, N) ;
         value  RHO, N; integer N; real RHO;
         begin
              real  SUM ; integer I ;
              if  N = 0  then  POINT ← 1
              else
                   begin
                        SUM ← POINT (RHO, N-1);
                        if  RHO ≧ 1 then
                             for  I ← 1, I + 1  while I↑2 ≦ RHO do
                                  SUM ← SUM + 2 *
                                    POINT (RHO - I↑2, N-1) ;
                        POINT ← SUM
                   end
         end POINT ( ) ;
```

A9. Write an ALGOL program to find a "stable" set of marraiges for a given group of N boys and N girls. Each boy has ordered the girls according to his preferences, and you are given the results as an array BOY in which

$BOY_{i,k}$ = (the number of the girl who is the kth choice of boy i).

Similarly, the girls have indicated their preferences in an array GIRL, in which

$GIRL_{i,k}$ = (the number of the boy who is the kth choice of girl i).

A set of marriages is unstable if a man and woman exist who are not married to each other but prefer each other to their actual mates. If there are no such discontent couples, then the set of marriages is stable.

One possible algorithm for solving this problem is contained in the following article:

Gale, D. and Shapley, L. S., College Admissions and the Stability of Marriages, American Mathematical Monthly, 69 (January 1962), pp. 9-15.

A10. A painting has been p~oduced in a rectangular frame with the use of three colors, or of any color that can be obtained by mixtures of these colors; call these colors red, yellow, and blue. We consider the painting divided into unit squares with N columns and M rows. You are given three matrices R, B, and Y; Rij is a real number indicating the amount (in some units) of red points in the square (i,j); similarly, Bij and Yij indicate the amounts ot blue and yellow paint, respectively, in square (i,j). Define a square to be reddish if it has more red paint than yellow and blue together, similarly blueish and yellowish.

Compute the number REDDISH, the proportion of the picture that is reddish, i.e., the number of reddish squares divided by the total number of squares. Similarly compute BLUEISH and YELLOWISH.

If red paint costs 2 cents a unit, blue 3 cents, and yellow 4 cents and if there is an additional charge of 50 cents for each different mixture used, compute also the total cost of the paint in dollars. You can assume that two squares were painted with the same mixture if the ratio of red paint to blue paint to yellow paint used in each square is the same (to within a tolerance determined by the accuracy of representation of real numbers in the machine). A pure red or blue or yellow does not count as a mixture.

A11. Given a painting as represented in problem 10 and the coordinates (X, Y) of a particular square, compute the number of squares painted using the same color of paint (i.e., the same pure color, red or blue or yellow, or the same mixture as defined in problem 10) as in square (X, Y) and connected to it by a chain of squares of the same color. A chain of squares is obtained by any combination of horizontal and vertical moves, but not diagonal moves.

B. Geometry Problems.

B1. Given a circle C defined by the triplet of values (X, Y, R) with R the radius and X, Y the coordinates of the center, and a point P defined by coordinates (U, V). Determine whether or not P is inside C, and set a Boolean variable INSIDE accordingly.

You are given two triangles A and B defined by the coordinates of their vertices:

A defined by (X_1, Y_1), (X_2, Y_2), and (X_3, Y_3) ;

B defined by (X_4, Y_4), (X_5, Y_5), and (X_6, Y_6) .

B2. Compute the area of the triangle A.

B3. Given two triangles A and B defined as above, determine whether A can be fitted inside B or made to coincide with B, using only translation and rotation.

B4. Given two circles C_1 and C_2, each defined by triplets (X_1,Y_1,R_1), and (X_2,Y_2,R_2) as in problem B1, compute the number of common tangents.

B5. Given the coordinates (X_1, Y_1), ..., (X_N, Y_N) of N vertices of a polygon, compute its area.

B6. Given N circles C_1, C_2, ..., C_N defined by triplets (X_1, Y_1, R_1), ..., (X_N, Y_N, R_N) as in problem B4, compute the area of the smallest square which encloses all N circles.

B7. Given four points $P_i = (X_i, Y_i)$, i = 1, 2, 3, 4, calculate the coefficients M and B for the equation: $y = MX + B$ of the line joining the midpoint of segment P_1P_2 with the midpoint of segment P_3P_4. Set the Boolean variables UNIQUE and SOLUTION as follows:

SOLUTION = <u>true</u> if there exists an equation of the form $y = MX + B$, <u>false</u> otherwise.

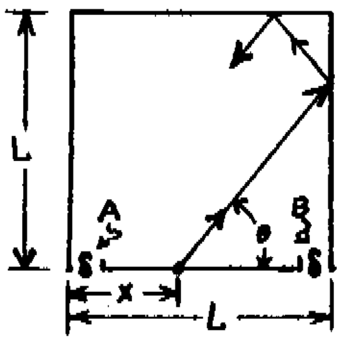
UNIQUE = <u>true</u> if SOLUTION = <u>true</u> and the computed M and B values are unique, <u>false</u> otherwise.

B8. Given two squares A and B defined by the coordinates of their vertices, compute their common area.

B9. Given a square defined by the coordinates of two diagonally opposite vertices, and given the coordinates of a point P, determine whether P lies inside the square. Set a Boolean variable PINSIDE to true if P lies inside, false otherwise.

B10. Assume a sequence of squares S_0, S_1, ..., S_k, ... are constructed such that each S_k is centered on the origin and has area $(1 + k)A^2$, for $k \geq 0$. Given a value for A and for the coordinates (X, Y) of a point P, compute the smallest k such that P is inside S_k.

B11. You are given a square of side L with two gaps A and B in the base, each of length δ. Assume a particle is fired from a point on the base, x units from the left side, at an angle $\pi > \theta > 0$. The particle rebounds elastically from the walls. Compute the distance, d, traveled by the particle before it exits through one of the gaps. If the particle will never exit through one of the gaps, set d to -1. Hint: Consider space filled by stacks of identical boxes with perfectly ethereal walls.



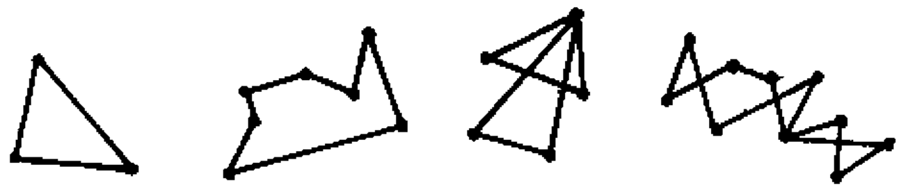
B12. Parts (a) through (e) which follow form a unified sequence of problems. In general, the solution to each part is useful as a subroutine in the parts which follow it in sequence. Before starting the problems, we give some definitions:

Two line segments intersect if they have exactly one point in common; they overlap if they have a line segment in common. A line segment always has non-zero length, so that a point is not a segment.

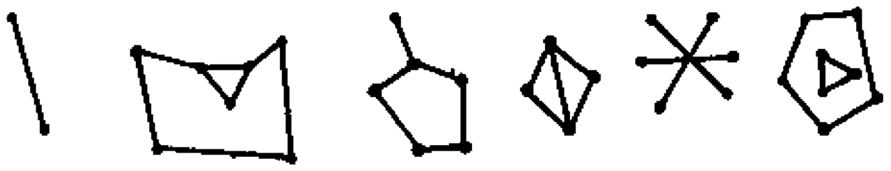
A polygon (or n-gon) is a set of n line segments (i.e., sides) such that one can trace a closed path over segments which (1) passes through each endpoint half as many times as the endpoint appears in a list of the segments, and which (2) passes along each segment exactly once.

Examples:

Polygons



Non-polygons



B12. (continued)

(a) Given a line segment L_{ab} defined by the coordinates (X_a, Y_a), (X_b, Y_b) of its endpoints, determine whether the point $P = (X_P, Y_P)$ lies <u>on</u> the segment (between a and b) and set a Boolean variable ONLINE accordingly.

(b) Given two line segments L_1 and L_2 defined by the coordinates of their endpoints, compute the coordinates (X_P, Y_P) of their point of <u>intersection</u>, and set the Boolean variable INTERSECT to be <u>true</u>. <u>If, however</u>, L_1 and L_2 do not intersect (within their <u>lengths</u>), then set INTERSECT to <u>false</u> and X_P and Y_P to 0.

Assume that the two line segments have <u>at</u> <u>most</u> one point in common, i.e., they do not <u>overlap</u>.

(c) You are given J line segments $L_1, \ldots, L_J$, represented by the coordinates of their endpoints; thus for each $i = 1, \ldots, J$, L_i is represented by $((X_{1i}, Y_{1i}), (X_{2i}, Y_{2i}))$. Find all points P_{ab} of intersection of pairs of segments L_a, L_b from the given set. Your result should include the number m of such intersections and a list of their coordinates, in the following order:

B12. (cont'd) (1) If $a < c$, then P_{ab} is listed before P_{cd}.

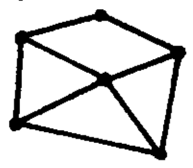
(2) If $b < d$, then P_{ab} is listed before P_{ad}.

As in Part (b), assume that no segments overlap.

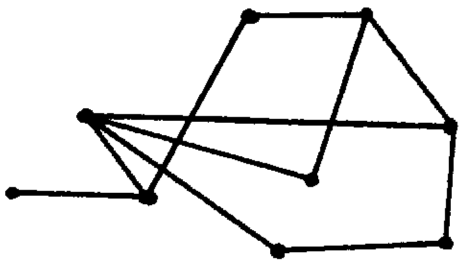
(d) Given J line segments $L_1, \ldots, L_J$ represented as in Part (c), determine whether they form a polygon (N-gon) when all N segments are included, and set a Boolean variable ISPOLYGON accordingly. To determine whether the segments do form a polygon (according to the definition given earlier), you must traverse the segments in some order. If the segments do form a polygon, your program must not only set ISPOLYGON to <u>true</u> but also reorder the segments to correspond to the order in which you traced them. It should also exchange endpoints of segments in the reordered list $L_1, \ldots, L_J$ if necessary so that the second endpoint of each segment L_k coincides with the first endpoint of the next segment L_{k+1}.

(e) Given J line segments as in Part (d), compute N, the greatest integer such that an N-gon can be constructed from some subset of the given segments. Thus, not all given segments need be used, and $0 \leq N \leq J$.

Examples:



J = 9
N = 6



J = 11
N = 7

B13. "The Triangle Problem":

You are given N line segments $L_1, \ldots, L_N$ defined by the coordinates of their endpoints. Find the first triangle, if any, which can be formed from these segments. We define "first" by the ordering:

(L_a, L_b, L_c) comes before (L_d, L_e, L_f)

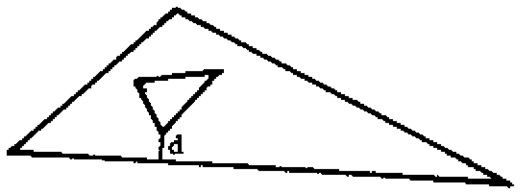
if $a < d$

or if $a = d$ and $b < e$

or if $a = d$ and $b = e$ and $c < f$.

If no triangles can be formed, set a Boolean variable NONE to <u>true</u>.

B14. You are given the vertices of two triangles A and B, such that triangle A encloses triangle B. Compute d, the minimum distance from any point on A to any point on B.



B15. Given a non-reentrant polygon of N sides defined by the N vertices $(X_1, Y_1), \ldots, (X_N, Y_N)$ and a point $P = (X_P, Y_P)$, determine whether P is inside or outside the polygon.

B16. Given a non-reentrant polygon of N sides defined as in problem B15, tag those of its vertices which are also the vertices of the convex hull of the polygon.

C. Representation Problems.

C1. Write programs to add, subtract, multiply, and divide Roman Numerals. A Roman Numeral can be represented by a sequence $c_k, c_{k-1}, \ldots, c_1$ of codes, where each c_i will be 1, 2, 3, 4, 5, or 6 to represent I, V, X, L, C, or M, respectively. The code c_1 is the lowest-order (i.e., the right-most character in the Numeral.)

C2. You are given two ordered triples (M_1, D_1, Y_1) and (M_2, D_2, Y_2) which represent calendar dates by (month, day, year) numbers. Compute the total number of days included by these two dates. If, however, either or both given triples do not represent legitimate expressions for dates, set the number of days spanned to -1.

C3. Find all convex polygons of area 8 whose sides are either parallel to the coordinate axes and have integer lengths, or are 45° diagonals with lengths equal to an integer multiple of $\sqrt{2}$. Polygons differing only in translations from the origin are to be considered the same polygon.

C4. Consider an abstract network composed of N nodes connected together by lines. If the nodes are numbered 1 through N, then the network is defined by its Boolean <u>connection</u> matrix C, where:

```
    C_ij = (if there is a line connecting node i directly to node j
            then true,
            else false),
```

for all (i, j) = 1, 2, ..., N. Write a program which, given such a connection matrix for a network, computes the corresponding Boolean _path_ matrix P, defined by:

P_{ij} = (_if_ there is a path, through any number of intervening nodes, from node i to node j _then_ _true_, _else_ _false_).

C5. A maze can be represented as an abstract network specified by a connection matrix as defined in problem C4. Assume you are given the connection matrix for a maze as well as the entrance node number E and goal node number G. Find the shortest path through the maze from E to G.

C6. A directed graph can be thought of as a set of nodes connected by unidirectional lines or arrows. The Boolean connection matrix C for an N-node directed graph can be defined as:

C_{ij} = (if there is an arrow directed from node i to node j _then_ _true_, _else_ _false_)

for all i, j = 1, 2, ..., N. Write a program which determines whether or not a given directed graph, defined by its connection matrix, contains any closed loops. If there are any loops, compute a new connection matrix which contains only those arrows of the original graph which are contained in at least one of the loops.

C7. A topological ordering of the nodes of a directed graph is an ordering in which node i precedes node j if there is a path through the network from node i to node j. Write a program to find a topological ordering for the nodes of an N-node directed graph specified by its connection matrix, and set ORDERED to _true_. If, however, there are one or more loops in the graph then a topological order cannot be found and you should set ORDER to _false_. Note: Topological ordering is the basis of the "PERT" (Program Evaluation and Review Technique) for management of complex industrial activities.

C8. An abstract group of order N is defined by its "multiplication table". If the elements of the group are $e_1, \ldots, e_N$, then the multiplication table can be written as a matrix M, where $M_{ij} = p$ if and only if $e_i e_j = e_p$. Write a program which determines whether or not a given NxN matrix M is the multiplication table for a group of order N, by checking the four group postulates:

(1) The group must be closed under the operation and the operation must be unique.

(2) There must be a left identity element in the group.

(3) For every element in the group there must be a left-inverse in the group.

(4) The operation must be associative.

Postulate (1) is guaranteed by the M matrix which will be given to you; if the other three hold also, set ISGROUP to <u>true</u>.

C9. The commutator of any two elements e_i and e_j of an abstract group G is the element

$$e_i^{-1} e_j^{-1} e_i e_j .$$

Given a group defined as in problem C8 by its M matrix, construct the multiplication table products of all commutators from G. The new multiplication table itself defines a group G' which is therefor a subgroup of G. Determine the order of G'.

C10. Write a program which, given the multiplication table for a group G, computes the multiplication table for a subgroup G' of G, if such a subgroup exists.

D. Numerical Computation Problems.

D1. Find a zero of a given function by one of the standard methods:

(a) bisection.

(b) regula falsi.

(c) Newton's method (which requires a method for calculating the derivative).

D2. Given N, count the number of primes $< N$.

D3. Solve the general cubic equation:

$$Ax^3 + Bx^2 + Cx + D = 0, \text{ where } A \neq 0 .$$

D4. Solve N simultaneous linear equations.

D5. Perform numerical integration using the trapezoidal formula or Simpson's rule.

D7. Assuming there is no subroutine available, program the exponentiation operation X^n in one of the following cases:

(a) n an integer

(b) n a real number

(c) n real or integer, using a given Boolean variable ISREAL to determine the type.

(d) As in (c), but optimized for the most common cases: n = positive integer $\leqq 5$.

(e) As in (d), but minimizing number of multiplications when n is an integer. Suggestion: Consider the expansion of n as a binary number.

D7. Program procedures (subroutines) for performing the operations of arithmetic on data of any of the following classes:

(a) Complex numbers, in polar or rectangular form.

(b) Double precision numbers.

(c) Extended floating point numbers.

(d) Polynomials.

(e) Piece-wise linear functions.

D8. Given an (unknown) function f(x) defined by a subroutine, find a (local) maximum of f to within a specified tolerance. Assume that there is a penalty associated with evaluations of f and hence minimize the number of evaluations.

D9. Write a program to compute a transcendental function - for example, $\log_{10}(X)$ - to specified accuracy for any value of X.

(a) Evaluate a best-fit polynomial approximation. For example, Hastings* gives coefficients $C_1, \ldots, C_9$ for which:

$$P(X) = 1/2 + C_1Y + C_3Y^3 + C_5Y^5 + C_7Y^7 + C_9Y^9$$

with $Y = \frac{X-\sqrt{10}}{X+\sqrt{10}}$ differs from $\log_{10}(X)$ by less than 2×10^{-7} for $1 \leqq X \leqq 10$. Evaluate the factored form of the polynomial.

(b) Given any $X > 0$, "reduce" it by factors of 10 or 1/10 until it lies in the range $1 \leqq X \leqq 10$, and then compute $\log_{10}(X)$ using part (a). If $X \leqq 0$, set an error flag.

---

* Hastings, C., Approximations for Digital Computers, Princetion University Press, 1955.

D10. Given integers n and c and real value x, evaluate $S(n, c, x)$ by the recursion relations:

(1) $n < 0$, any c: $S(n, c, x) = 0$.

(2) $n \geqq 0$, $c < 0$: $S(n, c, x) = (-1)^c x^{-c} S(n + c, -c, x)$.

(3) $n \geqq 0$, $c \geqq 0$: $S(n, c, x) = \sum_{k=0}^{n} \frac{(-1)^K}{K!} \cdot \frac{X^K}{(K + c)!}$

Evaluate the sum in factored form. Aside: for $n \geqq 0$ and $c > 0$, $S(n, c, x)$ is the nth partial sum of the power series for $X^{-c} J_c (2 \sqrt{X})$.

D11. Given a set of "experimental" data $X_1, \ldots, X_n$, compute statistical measures:

(a) Mean, Median, or Mode

(b) Frequency distribution

(c) Standard deviation

(d) Higher moments

D12. Compute as accurately as possible a value for the physical quantity α, called Modelung's constant, for a face-centered cubic ionic lattice:

$$\alpha = \sum_{p=-\infty}^{+\infty} \sum_{q=-\infty}^{+\infty} \sum_{r=-\infty}^{+\infty} \frac{(-1)^{p+q+r}}{p^2 + q^2 + r^2}$$

You will be given a fixed amount of computer time; since this series converges very slowly, a great deal of cleverness is called for.

3. COMPLETE EXAMPLE OF A TEACH PROBLEM

A. The Statement of the Problem:

TEACH17 - The Professor Lost in the Woods.

A Carnegie Tech mathematics professor got lost while walking in the woods one day. When he realized his predicament, he sat down by the nearest tree and decided upon the following algorithm for finding his way to one of the two roads which traversed the woods.

(1) He would walk in straight lines from tree to tree until he found one of the roads. The woods were so dense that he would be able to see the road only when he was very close to it.

(2) He had a piece of chalk in his pocket (of course). He would place a chalk mark on each tree he reached so as not to walk in a circle. This chalk mark could been seen from any direction.

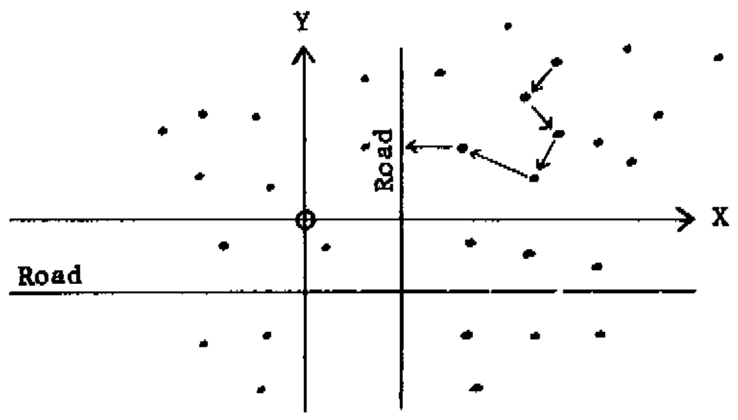
(3) Starting at the tree where he was sitting, and at each tree he reached, he would:

(a) put a chalk mark on the tree, and then

(b) look around and decide which unmarked tree was closest to his position; then he would walk in a straight line to this latter tree. If, however, a road came closer to him than any tree, then he would walk directly to the road at its nearest point.

(4) In deciding upon the nearest tree the professor could only be sure of distances with an accuracy of 1 percent. That is, if the distances of two trees differed by less than 1 percent of the larger of the two distances, then he judged them to be at equal distance. If it turned out that there was more than one tree at the closest distance (within the 1 percent error), he would decide among them by the following rule: he would take the one whose direction was most nearly North (which he could determine from the position of the sun). If two happened to make exactly equal angles with North, he would take the more easterly one (since he was right-handed).

Our professor was lost in a forest of ideal trees with perfectly straight and infinitely slender trunks, all perfectly perpendicular to a plane surface. He was ideal, too, with negligible dimensions.



Write an ALGOL program to follow the path of the professor. Assume a forest of N distinct trees whose (x,y) coordinates are specified by elements 1:N of two real arrays X and Y. The two roads run north-south and east-west, with coordinates given by x = NSROAD and y = EWROAD, respectively. The professor starts at tree K with coordinates (X [K], Y [K]). You will be given values of integers N and K, as well as real numbers NSROAD and EWROAD and real arrays X and Y. Assume that 1 ≦ K ≦ N ≦ 200. From this data you are to compute:

(1) The integer NUMBER, the number of trees which the professor reaches, not counting tree K where he starts.

(2) The contents of an integer array PATH whose successive elements are the subscripts of the trees which the professor reaches, in the order in which he reaches them. That is, he walks from:

```
(X [K], Y [K]) to (X [PATH [1]], Y[PATH [1]])
               to (X [PATH [2]], Y[PATH [2]]), etc.
and finally    to (X [PATH [NUMBER]], Y [PATH [NUMBER]]).
```

(3) A real variable DISTANCE, the total distance he walks to reach one of the roads.

To call TEACH17, use the following two cards immediately following your declarations:

```
SY    LIBRARY TEACH17
      GRADE: TEACH17 (<student number>, N,K,X,Y,NSROAD,EWROAD,
                      NUMBER, PATH, DISTANCE);
```

You can make the following (simplifying) assumptions:

(A) We will give you X and Y elements less than 10^4 in magnitude.

(B) No more than two trees will tie (within 1 percent) for the smallest distance.

B. CORRECT SOLUTION TO PROBLEM 17

```
          COMMENT  THIS IS A CORRECT SOLUTION TO  TEACH17   ....
                   THE PROFESSOR LOST IN THE WOODS     ;
11150   BEGIN
           REAL  DISTANCE, DMIN, ROADDISTANCE   ;
           INTEGER  I, N, NUMBER, S, T, Q   ;
11151      REAL ARRAY  X, Y [1:50]  ;
11162      INTEGER ARRAY  PATH[1 : 50]  ;

SY        LIBRARY  TEACH17
15013   NEXT: TEACH17 (200, N, S, X, Y, NUMBER, PATH, DISTANCE);

15036      T ← S   ;   COMMENT  THE PROFESSOR STARTS AT TREE S   ;
15040      NUMBER ← DISTANCE ← I ← 0   ;

        WALKAGAIN:
           BEGIN
15044         REAL DIST, DX, DY, TEST  ;    INTEGER  I    ;
15045      DMIN ← ⏨8   ;   COMMENT  CURRENT MINIMUM DISTANCE   ;
15047      Q ← 1  ;     COMMENT   BEGIN AT THE FIRST TREE  ... ;
15051      FOR I ← 1 STEP 1 UNTIL T-1, T+1 STEP 1 UNTIL N DO
              BEGIN
                 COMMENT   CONSIDER EACH TREE (EXCEPT T) IN TURN  ;
15114            DX ← X[I] - X[T]   ; DY ← Y[I] - Y[T]   ;
15134            DIST ← SQRT( DX↑2 + DY↑2) ;
15152            TEST ← ABS( X[Q] - X[T]) × DY
15163                 -( Y[Q] - Y[T] )×ABS(DX)  ;
15201            IF ABS( DIST - DMIN ) < .01×DIST
15204               ∨  ABS( DIST - DMIN ) < .01×DMIN
15217               THEN
                       BEGIN   COMMENT  DISTANCE OF TREE I AND TREE Q
                             ARE WITHIN 1 PERCENT  ;
15231                     IF TEST >0 ∨ ( TEST=0 ∧ X[I] > X[Q] )
15244                        THEN GO TO NEWTREE
15264                  END
15266            ELSE  IF DIST < DMIN THEN
15274   NEWTREE:       BEGIN   DMIN ← DIST ;  Q ← I  END   ;
              END  OF I LOOP   ;

15301      END  BLOCK   ;

15302      ROADDISTANCE ← ABS( IF ABS(X[T]) > ABS(Y[T]) THEN Y[T] ELSE X[T]);
15332      IF  ROADDISTANCE ≤ DMIN  THEN
              BEGIN
15336            DISTANCE ← DISTANCE + DMIN  ;
15341            NUMBER ← NUMBER + 1  ;
15344            X[T] ← ⏨10  ;   COMMENT  THE NEAT WAY TO CHALK THE TREE:
                    TRANSFORM IT OUT TO ''INFINITY''. SO IT WILL BE IGNORED.
                    A MORE GENERAL METHOD IS TO USE A BOOLEAN VECTOR 'CHALK';
15353            T ← PATH[NUMBER] ←  Q   ;
15361            GO TO WALKAGAIN ;
              END  STAGGER TO NEXT TREE

15362      ELSE  DISTANCE ← DISTANCE + ROADDISTANCE  ;
```

```
                     GO TO NEXT
          15366  END  SOLUTION TO PROBLEM 17
3197.  WORDS
                                                            00:00:47
```

(THIS IS THE END OF COMPILATION. PROGRAM EXECUTION FOLLOWS ON NEXT PAGE.)

```
 04 JUN 65
*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*

    1*      FOR DATA SET 1 YOU HAVE BEEN GIVEN     N=10   S=10

            YOUR PROFESSOR IS LOST IN THE FOLLOWING FOREST!

                                             TREE NUMBER      COORDINATES

                                                   1          (   79,   -11)
                                                   2          (   74,     4)
                                                   3          (   34,    49)
                                                   4          (   62,    20)
                                                   5          (   54,   -11)
                                                   6          (   23,    17)
                                                   7          (   59,    49)
                                                   8          (   12,    49)
                                                   9          (   34,    71)
                                                  10          (   78,     8)


            DATA SET  1 CORRECT

            TEACH ANSWERS ARE 'NUMBER'= 1  'DISTANCE'= 9.65685425 =+00
                                 PATH!    2   <END>

*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*

    1*      FOR DATA SET 2 YOU HAVE BEEN GIVEN     N=10   S= 3
                THE SAME GLOOMY FOREST AS THE LAST SET


            DATA SET  2 CORRECT

            TEACH ANSWERS ARE 'NUMBER'= 2  'DISTANCE'= 6.51126984 =+01
                                 PATH!    9,   8   <END>

*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*

    1*      FOR DATA SET 3 YOU HAVE BEEN GIVEN     N= 3   S= 3

            YOUR PROFESSOR IS LOST IN THE FOLLOWING FOREST!

                                             TREE NUMBER      COORDINATES

                                                   1          (   62,    20)
                                                   2          (   54,   -11)
                                                   3          (   23,    17)


            DATA SET  3 CORRECT

            TEACH ANSWERS ARE 'NUMBER'= 0  'DISTANCE'= 1.70000000 =+01
                                 PATH!         WALK DIRECTLY TO ROAD
```

```
*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/
  I*      FOR DATA SET 4 YOU HAVE BEEN GIVEN      N= 7    S= 7

          YOUR PROFESSOR IS LOST IN THE FOLLOWING FOREST:

                                         TREE NUMBER        COORDINATES

                                              1             (    54,   -11)
                                              2             (    23,    17)
                                              3             (    59,    49)
                                              4             (    12,    49)
                                              5             (    34,    71)
                                              6             (    78,     8)
                                              7             (    52,   -10)

          DATA SET  4 CORRECT

          TEACH ANSWERS ARE 'NUMBER'= 1  'DISTANCE'= 1.32360680 *+01
                            PATH:     1    <END>
*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/
  I*      FOR DATA SET 5 YOU HAVE BEEN GIVEN      N= 9    S= 1

          YOUR PROFESSOR IS LOST IN THE FOLLOWING FOREST:

                                         TREE NUMBER        COORDINATES

                                              1             (    34,    49)
                                              2             (    62,    20)
                                              3             (    54,   -11)
                                              4             (    23,    17)
                                              5             (    59,    49)
                                              6             (    12,    49)
                                              7             (    34,    71)
                                              8             (    78,     8)
                                              9             (    52,   -10)

          DATA SET  5 CORRECT

          TEACH ANSWERS ARE 'NUMBER'= 2  'DISTANCE'= 6.51126984 *+01
                            PATH:     7,    6    <END>

       STUDENT NUMBER 200 SCORES 5 OUT OF 5
       TIME USED:  00:01:03  PAGES USED:      8               11:20:04
```

C. INCORRECT SOLUTION TO PROBLEM 17

```
################################################################################

C OPER. YG01 04 JUN 65  11:24:18  AL3OL  PAGES:  10  TIME:   1  CARDS:       TAPE:
467


A:   06 AUG 64       I:   06 AUG 64       F:   13 MAY 65       00:00:02

                COMMENT  THIS IS AN I*N*C*O*R*R*E*C*T SOLUTION TO TEACH17;
       11150  BEGIN
                 REAL  DISTANCE, DMIN, ROADDISTANCE    ;
                 INTEGER  I, N, NUMBER, S, T, Q   ;
       11151     REAL ARRAY  X, Y [1:50]  ;
       11162     INTEGER ARRAY  PATH[1 : 50]  ;

SY              LIBRARY  TEACH17
       15013  NEXT: TEACH17 (200, N, S, X, Y, NUMBER, PATH, DISTANCE);

       15036     T ← S    ;   COMMENT  THE PROFESSOR STARTS AT TREE S    ;
       15040     NUMBER ← I ← 0  ;
                 COMMENT  **** HE DIDN'T INITIALIZE DISTANCE TO 0  ///  ;

              WALKAGAIN:
                 BEGIN
       15043       REAL DIST, DX, DY, TEST  ;     INTEGER  I    ;
       15044     DMIN ← @8   ;   COMMENT  CURRENT MINIMUM DISTANCE    ;
       15046     Q ← 1  ;     COMMENT   BEGIN AT THE FIRST TREE  ... ;
       15050     FOR I ← 1 STEP 1 UNTIL T-1, T+1 STEP 1 UNTIL N DO
                    BEGIN
                       COMMENT   CONSIDER EACH TREE (EXCEPT T) IN TURN  ;
       15113           DX ← X[I] - X[T]    ; DY ← Y[I] - Y[T]    ;
       15133           DIST ← SQRT( DX↑2 + DY↑2) ;
       15151           TEST ← ABS( X[Q] - X[T]) * DY
       15162                 -( Y[Q] - Y[T] )*ABS(DX)  ;
       15200           IF ABS( DIST - DMIN ) < .01*DIST
       15203              ∨  ABS( DIST - DMIN ) < .01*DMIN
       15216              THEN
                             BEGIN   COMMENT  DISTANCE OF TREE I AND TREE Q
                                   ARE WITHIN 1 PERCENT  ;
       15230                    IF TEST >0 ∨ ( TEST=0 ∧ X[I] > X[Q] )
       15243                       THEN GO TO NEWTREE
       15263                 END
       15265           ELSE  IF DIST < DMIN THEN
       15273  NEWTREE:       BEGIN   DMIN ← DIST ;  Q ← I  END    ;
                    END  OF I LOOP    ;

       15300     END  BLOCK   ;

       15301     ROADDISTANCE ← ABS( IF ABS(X[Q]) > ABS(Y[Q]) THEN Y[Q] ELSE X[Q]);
                    COMMENT  ****  THE SUBSCRIPTS SHOULD HAVE BEEN T IN THE ABOVE ;
       15331     IF  ROADDISTANCE ≤ DMIN  THEN
                    BEGIN
       15335           DISTANCE ← DISTANCE + DMIN  ;
       15340           NUMBER ← NUMBER + 1  ;
```

```
        15343            X[T] ← ₁₀10  ;   COMMENT  THE NEAT WAY TO CHALK THE TREE;
                            TRANSFORM IT OUT TO ''INFINITY'', SO IT WILL BE IGNORED.
                            A MORE GENERAL METHOD IS TO USE A BOOLEAN VECTOR 'CHALK';
        15352            T ← PATH[NUMBER] ←  Q   ;
        15360            GO TO WALKAGAIN ;
                      END  STAGGER TO NEXT TREE

        15361       ELSE  DISTANCE ← DISTANCE + ROADDISTANCE  ;
                    GO TO NEXT
        15365   END  SOLUTION TO PROBLEM 17
  3196.  WORDS                                                        00:00:29
```

```
 04 JUN 65
*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/

    !*     FOR DATA SET 1 YOU HAVE BEEN GIVEN     N=10    S=10

           YOUR PROFESSOR IS LOST IN THE FOLLOWING FOREST!

                                                TREE NUMBER      COORDINATES

                                                      1          (    79,   -11)
                                                      2          (    74,     4)
                                                      3          (    34,    49)
                                                      4          (    62,    20)
                                                      5          (    54,   -11)
                                                      6          (    23,    17)
                                                      7          (    59,    49)
                                                      8          (    12,    49)
                                                      9          (    34,    71)
                                                     10          (    78,     8)

           DATA SET  1 INCORRECT

                                 YOUR ANSWER, 'NUMBER'=   0, IS INCORRECT
                                 DID YOU FORGET TO INITIALIZE DISTANCE
                                 THE PATH WHICH YOU TRACED OUT IS :
                                 PATH! NONE    <END>

           TEACH ANSWERS ARE 'NUMBER'= 1  'DISTANCE'= 9.65685425 =+00
                                 PATH!    2    <END>

*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/

    !*     FOR DATA SET 2 YOU HAVE BEEN GIVEN     N=10    S= 3
               THE SAME GLOOMY FOREST AS THE LAST SET


           DATA SET  2 INCORRECT

                                 YOUR ANSWER, 'NUMBER'=   1, IS INCORRECT
                                 DID YOU FORGET TO INITIALIZE DISTANCE
                                 THE PATH WHICH YOU TRACED OUT IS :
                                 PATH!    9,NONE    <END>

           TEACH ANSWERS ARE 'NUMBER'= 2  'DISTANCE'= 6.51126984 =+01
                                 PATH!    9,    8    <END>

*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/

    !*     FOR DATA SET 3 YOU HAVE BEEN GIVEN     N= 3    S= 3

           YOUR PROFESSOR IS LOST IN THE FOLLOWING FOREST!

                                                TREE NUMBER      COORDINATES
```

```
                                                        1          (   62,    20)
                                                        2          (   54,   -11)
                                                        3          (   23,    17)


          DATA SET  3 INCORRECT

                           DID YOU FORGET TO INITIALIZE DISTANCE
                           THE PATH WHICH YOU TRACED OUT IS :
                           PATH:          WALK DIRECTLY TO ROAD

          TEACH ANSWERS ARE 'NUMBER'= 0  'DISTANCE'= 1.70000000 .+01
                           PATH:          WALK DIRECTLY TO ROAD

*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/

     !*     FOR DATA SET 4 YOU HAVE BEEN GIVEN     N= 7   S= 7

          YOUR PROFESSOR IS LOST IN THE FOLLOWING FOREST!

                                              TREE NUMBER     COORDINATES

                                                        1          (   54,   -11)
                                                        2          (   23,    17)
                                                        3          (   59,    49)
                                                        4          (   12,    49)
                                                        5          (   34,    71)
                                                        6          (   78,     8)
                                                        7          (   52,   -10)


          DATA SET  4 INCORRECT

                           DID YOU FORGET TO INITIALIZE DISTANCE
                           THE PATH WHICH YOU TRACED OUT IS :
                           PATH:    1   <END>

          TEACH ANSWERS ARE 'NUMBER'= 1  'DISTANCE'= 1.32360680 .+01
                           PATH:    1   <END>

*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/

     !*     FOR DATA SET 5 YOU HAVE BEEN GIVEN     N= 9   S= 1

          YOUR PROFESSOR IS LOST IN THE FOLLOWING FOREST!

                                              TREE NUMBER     COORDINATES

                                                        1          (   34,    49)
                                                        2          (   62,    20)
                                                        3          (   54,   -11)
                                                        4          (   23,    17)
                                                        5          (   59,    49)
                                                        6          (   12,    49)
                                                        7          (   34,    71)
```

```
                                                    8          (   78,    8)
                                                    9          (   52,  -10)

     DATA SET  5 INCORRECT

                              YOUR ANSWER, 'NUMBER'=   1, IS INCORRECT
                              DID YOU FORGET TO INITIALIZE DISTANCE
                              THE PATH WHICH YOU TRACED OUT IS :
                              PATH:     7,NONE   <END>

     TEACH ANSWERS ARE 'NUMBER'= 2  'DISTANCE'= 6.51126984 .+01
                              PATH:     7,   6   <END>

   STUDENT NUMBER 200 SCORE 0 OUT OF 5
   TIME USED:  00:00:45  PAGES USED:      5              11:25:02
```

D. Listing of the TEACH17 Procedure

```
################################################################################
C OPER. YG01 04 JUN 65  11:20:05  ALGOL  PAGES:  10  TIME:   1  CARDS:      TAPE:
467
```

(The solution shown in C above has been modified here to cause the TEACH procedure itself to be printed as it is compiled into the student's program.)

```
  11150  BEGIN
            REAL  DISTANCE, DMIN, ROADDISTANCE     ;
            INTEGER  I, N, NUMBER, S, T, Q    ;
  11151     REAL ARRAY  X, Y [1:50]  ;
  11162     INTEGER ARRAY  PATH[1 : 50]  ;

           PRINT  LIBRARY,
           LIBRARY  TEACH17
           LIBRARY  RAND;
  11171  REAL PROCEDURE  RAND( R, T) ;
            VALUE  T ;  INTEGER  R ;  REAL  T  ;   COMMENT
               RANDOM NUMBER GENERATOR .
                  SETS R TO NEXT PSEUDO-RANDOM NUMBER IN INTERVAL 0->R<2097151
                  AND RAND() TO R MAPPED INTO INTERVAL  0-> RAND() < T   ;
  11177     BEGIN   INTEGER I    ;
  11225        I ← R ← R*1953125  ;   COMMENT I.E. R ← MOD(R*5↑9,2097152)   ;
  11235        RAND ← I* ( T/2097152)     ;
  11241     END RAND()   ;

         COMMENT   THIS  IS THE BEGINNING OF THE TEXT COMPILED FOR TEACH17 **;

         LABEL XQADRP,XQEXPO;LIBRARY PROCEDURE RUNERROR;BOOLEAN MLXPQZ;
         INTEGER OOO,OCOO;

  11246  PROCEDURE TEACH17 (STUDNO, N, S, X, Y, NUMBER, PATH, DISTANCE);
                   VALUE STUDNO; INTEGER STUDNO, N,S,NUMBER;  REAL DISTANCE;
                  REAL ARRAY X,Y;  INTEGER ARRAY PATH;
               COMMENT   THIS TEACH PROCEDURE WAS WRITTEN BY JOHN WHITE  ;
  11270  BEGIN     BOOLEAN  ISPRINT  ;
                  INTEGER I, EST , P, NO,NEXT, R ;REAL TOTD, A, B,LENGTH;
  11315           BOOLEAN ARRAY CORECT[0:4];     INTEGER ARRAY TRAIL[1:50];
               INTEGER E, IX, IY ;
  11333        PROCEDURE  SETX ;
  11340           BEGIN  IX ← IX + 1 ;  X[IX] ← E ;  IF IX=N THEN IX← 0 END ;
  11365        PROCEDURE  SETY ;
  11372           BEGIN  IY ← IY + 1 ;  Y[IY] ← E ;  IF IY=N THEN IY← 0 END ;

  11417        PROCEDURE SETDAT;
  11424              BEGIN SWITCH DATSET ← D1, D2, D3, D4, D5;
  11453                    ISPRINT ←  TRUE  ;
  11455                    GOTO DATSET[OOO];
  11457           D1:  P←0; N←10; S←10;        GOTO WOODS;
  11473           D2:  P←0; N←10 ;  S ← 3 ; ISPRINT ← FALSE ; GO TO WOODS ;
  11511           D3:  P ← N ← S ← 3 ;        GO TO WOODS ;
```

```
11523           D4:  P←4; N←7;  S←7;         GO TO WOODS;
11537           D5:  P←2; N←9; S←1;
11551        WOODS: I ← 0;
11553           FOR EST←79,74,34,62,54,23,59,12,34,78,52 DO
11616           BEGIN I←I+1; A←I-P; IF A>0 THEN X[A]←EST; IF A=N THEN
11646                                                     GOTO EX1  END;
11651        EX1:   I ← 0;
11653           FOR EST←-11,4,49,20,-11,17,49,49,71,8,-10 DO
11716           BEGIN I←I+1; A←I-P; IF A>0 THEN Y[A]←EST; IF A=N THEN
11746                                                     GOTO EXIT  END;
11751        EXIT:     X[N+1]←Y[N+1]←3.45⏨69;
11776        END;

        STR:
12002       IF OOO=0 THEN BEGIN A←↓(STUDNO/100);  B←STUDNO-100*A;
12016                           IF A<2 ∨ A>9 THEN GOTO ZORCH;
12036                           IF A<7 ∧ B>35 THEN GOTO ZORCH;
12056                           GOTO OUT;
12060                           ZORCH: PRINT(<'YOU HAVE NOT SUPPLIED TEACH4 ',
12075                           'WITH A VALID STUDENT NUMBER.  SORRY, BUT WE ',
12111                           'CANT RUN YOUR PROGRAM.',2E>); HALT;
                        END;
12124       SETDAT;
12125            I ← TOTD ← 0;  IF X[S]=0 ∨ Y[S]=0 THEN GOTO ROAD;
            HUNT:

12164       BEGIN INTEGER XT,YT,CCT,XCCT, C1PCT,I;REAL DCCT,DC1PCT,XDCCT,TD,Q;
12165                      DCCT←⏨50; XT←X[S]; YT←Y[S]; CCT←C1PCT←0;
12212       FOR I ← 1 STEP 1 UNTIL N DO
12235       BEGIN IF I=S ∨ X[I]>1⏨5 THEN GOTO NXTTRE;
12262            TD ← SQRT((XT-X[I])↑2+(YT-Y[I])↑2));
12312       IF TD<DCCT THEN BEGIN XCCT←CCT; XDCCT←DCCT; CCT←I; DCCT←TD;
12327                           IF XDCCT-DCCT < .01*XDCCT THEN BEGIN C1PCT←XCCT;
12342                                                                  DC1PCT←XDCCT;
                                                                     END
12344                                                           ELSE C1PCT←0;
12347       GO NXTTRE;
                          END;
12351            IF C1PCT≠0 THEN GOTO NXTTRE
12356                          ELSE IF TD-DCCT < .01*TD  THEN BEGIN C1PCT←I;
12374                                                                  DC1PCT←TD;
                                                                   END;
12376       NXTTRE: END     ;
12377       IF CCT≠0 THEN BEGIN
12404       IF C1PCT=0 THEN BEGIN Q←CCT; GOTO SORT END;
12414       A←(Y[CCT]-YT)*ABS(X[C1PCT]-XT);  B←(Y[C1PCT]-YT)*ABS(X[CCT]-XT);
12450                            Q ←    IF ABS(A-B)->1⏨-8*(A+B+ABS(A-B)) THEN
12467                            ( IF X[CCT]>X[C1PCT] THEN CCT ELSE C1PCT   )
12504                              ELSE IF A>B THEN CCT ELSE C1PCT;
12517        SORT:  LENGTH←SQRT((X[Q]-XT)↑2+(Y[Q]-YT)↑2);  NEXT←Q;
                         END
12551             ELSE NEXT←0;
12554       END OF HUNT BLOCK;
```

```
12555  FOUND: A ← ABS(0 + X[S]);  B ← ABS(Y[S] + 0);
12601      IF (A<LENGTH) ∨ (B<LENGTH) ∨ (NEXT=0) THEN GOTO ROAD;
12630              X[S]←1.6; I←I+1;
12645              S←NEXT; TRAIL[I]←S; TOTD←TOTD+LENGTH;  GOTO HUNT;
12664      ROAD: NO←I;  TOTD←TOTD+ IF A<B THEN A ELSE B    ;

                         COMMENT     COMPARE (1) NO TO NUMBER
                                             (2) TRAIL TO PATH  (N)
                                             (3) TOTD TO DISTANCE  ;

12700      CORECT[1]← NO=NUMBER;
12713      CORECT[2]←TRUE;  FOR P←1 STEP 1 UNTIL NO DO CORECT[2]←CORECT[2]∧
12744                                                 PATH[P]=TRAIL[P];
12760      CORECT[3]← ABS(TOTD-DISTANCE) <1.-8*TOTD;
13000      CORECT[4]←TRUE;  IF 0<NUMBER ∧ NUMBER<NO THEN
13027                          FOR P←1 STEP 1 UNTIL NUMBER DO CORECT[4]←
13055                              CORECT[4]∧PATH[P]=TRAIL[P];
13071      CORECT[4]←¬CORECT[4];
13077     CORECT [0] ← CORECT [1] ∧ CORECT [2] ∧ CORECT [3];
13111      NAME(OOO); IF ¬CORECT[0] THEN PRINT(<   12C,'DATA SET ',2D,
13136                                           ' INCORRECT',2E>)
13143            ELSE BEGIN OCOO←OCOO+1; PRINT(<12C,'DATA SET ',2D,
13163                               ' CORRECT',2E>);  GOTO PANS;
                    END;
13173      IF NUMBER<0 THEN PRINT(<30C,'DID YOU FORGET TO INITIALIZE ',Q,
13221                                'NUMBER',Q,E>);
13230      IF ¬CORECT[1] THEN BEGIN NAME(NUMBER); PRINT(<30C,'YOUR ANSWER, ',
13257                               Q,'NUMBER',Q,'=',-3D,', IS INCORRECT'>,
13274                               →$(IF CORECT[4] THEN 1 ELSE 0)$
13302                               <' (NOT ENOUGH TREES IN PATH)'>,<E>);
13327                             IF CORECT[4] THEN PRINT(<40C,'ALSO, THAT ' ,
13343                             'PORTION OF THE PATH WHICH YOU DID TRACE ',
13356                             'OUT IS INCORRECT', E>);
                            END;
13366      IF DISTANCE <0 THEN PRINT(<30C,'DID YOU FORGET TO INITIALIZE ',
13413                                'DISTANCE',E>)
13417         ELSE IF ¬CORECT[3] THEN BEGIN NAME(DISTANCE);
13437                               PRINT(<30C,'YOUR ANSWER, ',Q,'DISTANCE',
13455                               Q,'=',-1D.8ZL,' IS INCORRECT',E>)
13467                                  END;
13471      PRINT(<30C,'THE PATH WHICH YOU TRACED OUT IS :',E,30C,'PATH: '>);
13517         EST ← IF NO< NUMBER THEN NUMBER ELSE NO;
13534         P ← IF 15 < EST THEN 15 ELSE EST;
13545            FOR I ← 1 STEP 1 UNTIL P DO
                    IF PATH[I] = -1000
13566               THEN BEGIN NAME('NONE'); PRINT(<4A,','>)  END
13615               ELSE BEGIN NAME(PATH[I]);  PRINT(<-3D,','>)  END;
13642      IF  P = 0 THEN PRINT(<35C,10B,'WALK DIRECTLY TO ROAD',52B,2E>)
13665                 ELSE PRINT(<1L,3B,'<END>',2E>);
13704      PANS: NAME(NO,TOTD,I→NO(TRAIL[I]));
13734         PRINT(<12C,'TEACH ANSWERS ARE ',Q,'NUMBER',Q,'=',2D,2B,Q,
13761                 'DISTANCE',Q,'=',-1D.8ZL,E,30C,'PATH: '>,
13775                 →NO<-3D,','>);
14014      IF NO = 0 THEN PRINT(<35C,10B,'WALK DIRECTLY TO ROAD',52B,2E>)
```

```
       14037                 ELSE PRINT(<1L,3B,'<END>',2E>);
                   OUT:
       14056            IF OOO=5 THEN GOTO FIN;
       14064            PRINT(+60<'*/'>,<2E>)  ; OOO+OOO+1;  SETDAT;
       14111       NUMBER+DISTANCE+-1000; FOR I+1 STEP 1 UNTIL 20 DO PATH[I]+-1000;
       14150       NAME(OOO,N,S);
       14172       PRINT(<5C,'(+',12C,'FOR DATA SET ',1D,' YOU HAVE BEEN GIVEN    ',
       14217             'N=',2D,'   S=',2D,E>);
       14231       IF ISPRINT THEN
       14233                     BEGIN  NAME( I+N(I,X[I],Y[I])); PRINT(<120B,E>);
       14302                     PRINT(<12C,'YOUR PROFESSOR IS LOST IN THE FOLLOWING'
       14321                           ,' FOREST:',2E,40C,'TREE NUMBER      ',
       14334                           'COORDINATES',2E>,+N<45C,2D,56C,'(',-4D,',',
       14364                            -4D,')',E>,<2E>);  END
       14375                 ELSE PRINT(<16C,'THE SAME GLOOMY FOREST AS THE LAST SET'
       14415                    ,3E>)  ;
       14421       GOTO TCHND;
       14423       FIN: NAME(STUDNO,OCOO,OOO);
       14441       PRINT(<73C,'STUDENT NUMBER ',3Z,' SCORES ',2D,' OUT OF ',2D,E>);
       14467       HALT;
                   TCHND:
       14470   END OF TEACH17;


                        COMMENT  ***  THE FOLLOWING STATEMENTS RECOVER CONTROL
                               FOR TEACH IN CASE OF EXPONENT OVERFLOW AND ADROPS ;
       14474   PROCEDURE ADDRESS.MESSAGE; BEGIN INTEGER T;
WH     14502             CLA  2   (33,5)           GET ADDRESS
       14503             STI  0   T;               SAVE IN TEMP
AL     14504   NAME(T);
       14514            PRINT(<12C,'THE ADDRESS OF THE COMMAND TO BE EXECUTED ',
       14535   'FOLLOWING THE COMMAND WHICH CAUSED THE ERROR IS',AD,2E>); END;
       14562   MLXPQZ+FALSE;OOO+OCOO+0; GOTO BCWLXZ;
       14571   XQADRP:  PRINT(<12C,'YOUR PROGRAM HAS CAUSED THE G-21 TO ATTEMPT ',
       14612   'TO EXECUTE AN ILLEGAL COMMAND. EITHER SOME FORMAL AND ACTUAL ',
       14633   'PARAMETERS',E,12C,'ARE NOT IN CORRESPONDENCE OR A SUBSCRIPT IS OUT '
       14655   ,'OF BOUNDS.',2E>);    ADDRESS.MESSAGE; GOTO BCWLXZ;
       14670   XQEXPO:  PRINT(<12C,'YOUR PROGRAM HAS CREATED AN EXPONENT OVERFLOW.',
       14712   '  PLEASE CHECK FOR DIVISION BY A VERY SMALL NUMBER',E,12C,
       14732    'OR FOR MULTIPLICATION OF TWO VERY LARGE NUMBERS.',2E>);
       14752   ADDRESS.MESSAGE;
       14753   BCWLXZ:  RUNERROR(XQADRP,'ADRP',MLXPQZ);
       14773            RUNERROR(XQEXPO,'EXPO',MLXPQZ);
       15013   NEXT: TEACH17 (200, N, S, X, Y, NUMBER, PATH, DISTANCE);
       15036      T + S    ;   COMMENT  THE PROFESSOR STARTS AT TREE S    ;
       15040      NUMBER + I + 0  ;
                  COMMENT  **** HE DIDN'T INITIALIZE DISTANCE TO 0  ///  ;
               WALKAGAIN:
                  BEGIN
       15043         REAL DIST, DX, DY, TEST  ;    INTEGER  I   ;
       15044      DMIN + .8   ;   COMMENT  CURRENT MINIMUM DISTANCE   ;
       15046      Q + 1  ;      COMMENT   BEGIN AT THE FIRST TREE  ... ;
       15050      FOR I + 1 STEP 1 UNTIL T-1, T+1 STEP 1 UNTIL N DO
                     BEGIN
                        COMMENT   CONSIDER EACH TREE (EXCEPT T) IN TURN  ;
```

```
        15113            DX ← X[I] - X[T]    ; DY ← Y[I] - Y[T]    ;
        15133            DIST ← SQRT( DX↑2 + DY↑2) ;
        15151            TEST ← ABS( X[Q] - X[T]) × DY
        15162                  -( Y[Q] - Y[T] )×ABS(DX)   ;
        15200            IF ABS( DIST - DMIN ) < .01×DIST
        15203               ∨  ABS( DIST - DMIN ) < .01×DMIN
        15216               THEN
                               BEGIN   COMMENT  DISTANCE OF TREE I AND TREE Q
                                     ARE WITHIN 1 PERCENT  ;
        15230                     IF TEST >0 ∨ ( TEST=0 ∧ X[I] > X[Q] )
        15243                        THEN GO TO NEWTREE
        15263                  END
        15265            ELSE  IF DIST < DMIN THEN
        15273   NEWTREE:       BEGIN   DMIN ← DIST ;  Q ← I  END    ;
                      END  OF I LOOP   ;

        15300      END  BLOCK   ;

        15301      ROADDISTANCE ← ABS( IF ABS(X[Q]) > ABS(Y[Q]) THEN Y[Q] ELSE X[Q]);
                      COMMENT  ****  THE SUBSCRIPTS SHOULD HAVE BEEN Q IN THE ABOVE ;
        15331      IF  ROADDISTANCE ¬< DMIN  THEN
                      BEGIN
        15335            DISTANCE ← DISTANCE + DMIN   ;
        15340            NUMBER ← NUMBER + 1  ;
        15343            X[T] ← ⏨10  ;   COMMENT  THE NEAT WAY TO CHALK THE TREE:
                            TRANSFORM IT OUT TO ''INFINITY''. SO IT WILL BE IGNORED.
                            A MORE GENERAL METHOD IS TO USE A BOOLEAN VECTOR 'CHALK';
        15352            T ← PATH[NUMBER] ←  Q    ;
        15360            GO TO WALKAGAIN ;
                      END  STAGGER TO NEXT TREE

        15361      ELSE  DISTANCE ← DISTANCE + ROADDISTANCE   ;
                   GO TO NEXT
        15365   END  SOLUTION TO PROBLEM 17
   3196.  WORDS                                                   00:00:37
```

4. EXAMINATION PROBLEMS CONCERNING PROGRAMMING

We list here some general classes of problems which convern programming in ALGOL - or any computer language - and are suitable for written examinations in an introductory programming course.

(1) Tell what a given ALGOL program does. Give values of specified variables at various intermediate points during execution.

(2) Draw a flow chart for a given ALGOL program.

(3) Write an ALGOL program from a given flow chart.

(4) Detect and correct errors of syntax in a given ALGOL program. Make any reasonable correction, without worrying about what the program is supposed to do - it is probably nonsense.

(5) Detect and correct "semantic" errors in a syntactically correct program which is intended to perform a specified task.

(6) Given a specified subset of the ALGOL language, indicate how the ALGOL constructions which have been omitted from the subset can be replaced by constructions in the subset. To clariby this, we will give examples both from ALGOL and from machine language.

Example 1: ALGOL Subset

LI'L ALGOL is a programming language similar to ALGOL in every respect, except that the following ALGOL constructions are <u>not</u> allowed:

<u>for</u> statements
switches
relations (except that = and > are permitted)
Boolean operators (i.e., ¬, ∧, ∨)
<u>else</u>

Each algorithm written in ALGOL can also be written in LI'L ALGOL.

For each of the following ALGOL constructions, write a LI'L ALGOL construction which has the same effect.

(a) <u>if</u> A = B <u>then</u> X ← X + 1 <u>else</u> B ← sin(X) ; NEXT:
(b) <u>if</u> C < D ∨ N ≠ 1 <u>then</u> <u>go</u> <u>to</u> SKIBO ; NEXT
(c) <u>for</u> S ← .3, T <u>step</u> 1 <u>until</u> 20 <u>do</u> X ; NEXT:
(d) <u>switch</u> G ← L1, L2, L3 ; <u>go</u> <u>to</u> G[I] ; NEXT:

Example 2: Machine Language Subset.

Suppose that you have a very simple computing machine which handles only integers and whose machine language consists of the following:

(1) Each command contains only an <u>op code</u> and an <u>address</u> A.

(2) There are only four possible op codes:

Op Code	Mnemonic	Meaning
CLA	CLear and Add	AC $\leftarrow$ MEM[A]
STO	STOre	MEM[A] $\leftarrow$ AC
CHS	CHange Sign	AC $\leftarrow$ -AC
SOT	Subtract One and Test	AC $\leftarrow$ -AC - 1 ; <u>if</u> AC < 0 <u>then</u> NC $\leftarrow$ A;

AC is a pseudo-ALGOL variable which represents the "ACcumulator". NC represents the "Next Command" register which contains the address of the next command to be executed. That is, CHS merely changes the sign of the accumulator AC. SOT simply subtracts 1 from the AC contents; if the result is less than 0, then the next command is taken from location A instead of from the location following the SOT command.

Here is a section of a program on your simple machine. Assume that the three names ZERO, Q, and REM stand for addresses of three data cells, and that cell ZERO always contains the number 0. Draw a primitive flow chart of the process performed by this program.

```
Location        Opcode  Address
  1000           CLA     REM
  1001           SOT     1002
  1002           SOT     1003
  1003           SOT     1012
  1004           STO     REM
  1005           CLA     Q
  1006           CHS
  1007           SOT     1008
  1008           CHS
  1009           STO     Q
  1010           CLA     ZERO
  1011           SOT     1000
  1012           (next segment)
```

Below is a list of short problems to be programmed in this simple machine language. The problems marked with asterisks are harder than the others. Commands may be placed into any cells; if additional temporary data cells are needed, call their addresses TEMP1, TEMP2, etc. The best solutions will be the shortest ones.

(a) <u>If</u> REM < 0 <u>then</u> REM ← -REM <u>else</u> Q ← -Q

(b) $\left\{\begin{array}{l}\text{0 if REM contains an odd number}\\ \text{1 if REM contains an even number.}\end{array}\right\}$ Here the cell REM contains a number $\geqq 0$.

*(c) Q ← max (Q, REM)

*(d) Perform general addition: REM + Q, leaving sum in AC. REM and Q have either sign.

PART V. BIBLIOGRAPHY

General Programming:

Arden, B. W., An Introduction to Digital Computing, Addison-Wesley, 1962.
Galler, B. A., The Language of Computers: An Introduction, McGraw-Hill, 1962.

Introduction to ALGOL:

Anderson, C., An Introduction to ALGOL 60, Addison-Wesley, 1964.
Bauer, Feliciana, and Samelson, Introduction to Algol, Prentice-Hall, 1964.
Dijkstra, E. W., A Primer of ALGOL 60 Programming, Academic Press, 1962.
McCracken, D. D., A Guide to ALGOL Programming, Wiley, 1962.

Other Computer Languages:

McCracken, D. D., A Guide to FORTRAN Programming, Wiley, 1961.
Organick, E., A FORTRAN Primer, Addison-Wesley, 1963.
Sherman, P. M., Programming and Coding Digital Computers, Wiley, 1963. This excellent book deals mostly with machine language.

Introductory Numerical Analysis:

Henrici, P., Elements of Numerical Analysis, Wiley, 1964.
McCracken, D. D. and Dorn, W. S., Numerical Methods and FORTRAN Programming, Wiley, 1964.
Pennington, R. H., Introductory Computer Methods and Numerical Analysis, MacMillan, 1965.

More advanced treatments of ALGOL:

Bottenbruch, H., "Structure and Use of ALGOL 60", J. Assoc. Comp. Mach. 9, (April 1962), 161-221.
Naur, P. (editor), "Revised Report on the Algorithmic Language ALGOL-60". Comm. Assoc. Comp. Mach. 6, (January 1963), 1-17.

Bottenbruch's tutorial article contains a particularly good treatment of procedures. The report edited by Naur is the basic official document defining the ALGOL language.

General Technical Journals:

Communications of the ACM, published by the Association for Computing Machinery, 211 East 43 Street, N.Y. 17, N.Y.
Journal of the ACM, same publisher as Communications above.
Computer Journal, published by the British Computer Society but available by subscription to members of the Association for Computing Machinery.

These journals are a fertile source of ideas and inspirations for programming problems. We should also point out that the Communications of the ACM contains an Algorithm section each month, in which are published a wide variety of algorithms for numerical and non-numerical tasks.