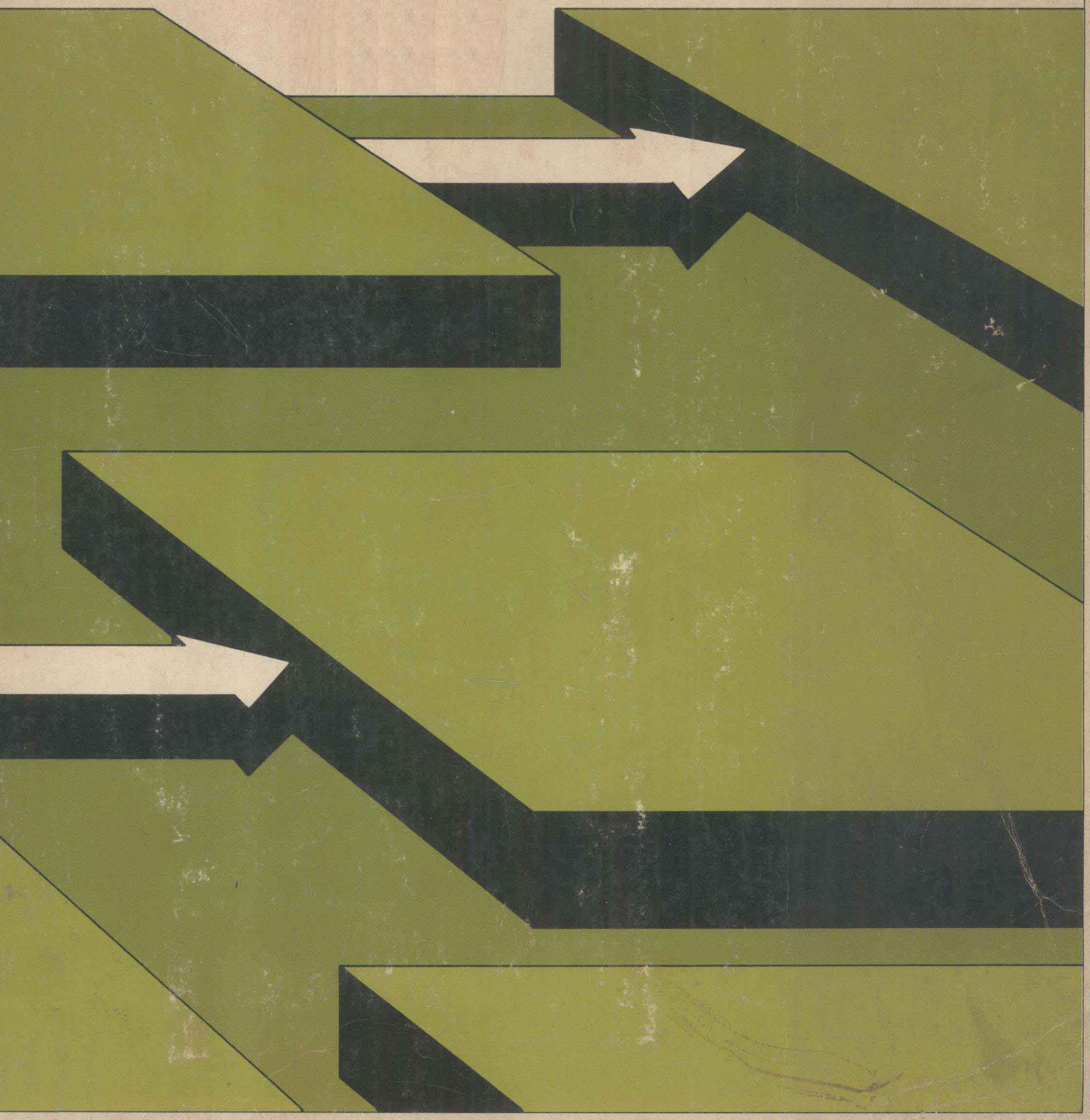
INTRODUCTION TO COMPUTERS,
STRUCTURED PROGRAMMING, AND APPLICATIONS

PASCAL Language Manual

STEPHEN CHERNICOFF



INTRODUCTION TO COMPUTERS, STRUCTURED PROGRAMMING, AND APPLICATIONS

PASCAL
Language Mannual

Stephen Chernicoff

L'homme n'est qu'un roseau, le plus faible
de la nature; mais c'est un roseau pensant.

Man is but a reed, the weakest in
nature; but he is a thinking reed.

--Blaise Pascal
(1623-1662)

Library of Congress Cataloging in Publication Data

Chernicoff, Stephen B
Pascal manual.

(Introduction to computers, structured programming, and applications)
Includes index.
1. PASCAL (Computer program language) I. Title. II. Series.
QA76.73.P2C43 001.6'424 77-19165
ISBN 0-574-21193-4

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

PREFACE

The programming language Pascal was introduced by Niklaus Wirth in the early 1970s to meet the need for an instructional language for the training of beginning programmers that would incorporate recent developments in programming language design. Its combination of "structured" control facilities, powerful data structures, and simplicity of expression make it especially suitable as a first language for students with no previous programming experience.

This Pascal primer is intended to be used in conjunction with the text Introduction to Computers, Structured Programming, and Applications, by C. W. Gear, referred to throughout as "the main text" or, simply, "the text." Corresponding chapters, sections, and example programs in the two books are correspondingly numbered, those in the Pascal volume being distinguished by underlining--that is, Section P7.2 corresponds to Section P7.2 of the text, Program A3.5 is the Pascal version of Program A3.5 of the text, and so on. The correspondence is not exact, however, since there are sections and programs in this book that have no counterparts in the main text. One unfortunate result is that the program numbers in this book are not strictly consecutive.

Part G of this book (General Introduction) consists of two chapters of basic information on how to code and run Pascal programs at a typical campus computer installation. Parts P (Programming) and A (Applications) parallel, in a general way, the structure of the corresponding modules of the main text. Because the emphasis in this volume is on the programming language itself, Part A is necessarily rather abbreviated, and consists entirely of Pascal versions of programs taken from a few selected chapters of the text. Part C bears no relation to Module C of the main text, but instead contains appendices summarizing various aspects of the Pascal language for quick reference.

The official definition of the Pascal programming language is contained in Wirth's Revised Pascal Report (1973). One of the aims of this book has been to present a complete description of Standard Pascal, as defined in that report. The few references to nonstandard features, or to details of a particular implementation, are generally to the Pascal 6000 dialect for the CDC 6000-series computers.

Finally, I wish to express my deepest gratitude and appreciation to my wife, Helen, for her untiring assistance, support, and advice in the preparation of this volume, and to Professor Robert Fabry of the University of California, Berkeley, whose aid was invaluable.

Stephen Chernicoff

January 1978

TABLE OF CONTENTS

G1 USING A COMPUTER IN PASCAL

A well-known children's number game runs as follows: Select your favorite number between 1 and 9, triple it, triple it again, and then multiply by the magic number 12345679. The result will be a number consisting of your favorite number repeated nine times. For instance, if the number you selected were 4, you would triple it to get 12, triple it again to get 36, and then multiply by the "magic number" to get 444444444.

Program G1.1 is a Pascal program to perform the calculation just described. The first line of the program,

```
program Magic (INPUT, OUTPUT);
```

is the program heading, and specifies, among other things, the name of the program. In this case we have named our program Magic, because it "magically" produces a long string of our favorite number. The heading is followed by an explanatory comment that has been inserted to help clarify what the program does for the benefit of the human reader. Next comes the line

```
var A, B, C, D: INTEGER;
```

which conveys information to the Pascal compiler about the variables used in the program. This line is called a declaration, and introduces four variables named A, B, C, and D. (The meaning of the word INTEGER here will become clear in Chapter P1.)

The remainder of Program G1.1--from the line containing the word begin to the line containing the word end--is called the body of the program, and describes the actions the program is to perform. Each action is represented by a statement. The first statement in Program G1.1 is

```
READLN(A)
```

which means "READ a LiNe of input to obtain a value for variable A." Our "favorite number" will be read in and assigned as the value of A. The next three statements perform the steps in the computation described earlier: set variable B to 3 times A, set C to 3 times B, and then set D to C times 12345679. (Notice that the asterisk * is used to represent the multiplication operation in Pascal, as in most other programming languages, and that the combination := represents the operation of assigning a value to a variable, corresponding to the left arrow ← in the main text.) Next come two WRITELN (WRITE a LiNe) statements to generate the program's printed output. The first WRITELN prints a line consisting of the message "Your favorite number was" followed by the value of variable A, the number originally input; the second prints the message "The result is" followed by the result calculated by the program and stored in variable D. Finally there is a line marking the end of the program.

- -

Program G1.1 Children's game

```
program Magic (INPUT, OUTPUT);

   (* Example Pascal program to generate a row of someone's favorite
      number, using the computation described in the text. *)

   var A, B, C, D: INTEGER;

   begin (* Magic *)
     READLN(A);
     B := 3 * A;
     C := 3 * B;
     D := C * 12345679;
     WRITELN('Your favorite number was ', A);
     WRITELN('The result is ', D)
     end (* Magic *).
```

Now that we have written a Pascal program, we must somehow enter it into the computer in order to get it run. Typically this means punching the program on computer cards or entering it from the keyboard of an online terminal. In either case, the format of the program on the input medium is free-form. A word or a number may not be split between two consecutive lines, but line boundaries are otherwise ignored: several statements may appear on a single line, or one statement may stretch over several lines. The information on a line may begin and end in any column, and extra blank columns may be inserted anywhere within a line (except in the middle of a word or number) to improve the program's readability. Similarly, whole blank lines may be inserted anywhere within a program. It is a good idea to make use of this freedom to help display the structure of your program to the human reader. In Program G1.1 we have inserted blank lines between the heading, the introductory comment, the variable declaration, and the program body, and have used horizontal indentation to accentuate the program's structure. As we encounter more complicated programs in the chapters to come, you will see how helpful the use of vertical spacing and horizontal indentation can be in clarifying the structure of the program. In your own programs, you needn't use exactly the same formatting conventions we use in this book, but you should adopt some consistent set of conventions to help reflect the structure of your program in its visual appearance.

Running a Pascal program on a computer is a two-stage process. The computer cannot execute Pascal programs directly: it only understands its own machine language, which is different for each particular type of computer. A program written in a machine-independent language such as Pascal must be compiled, or translated, into the computer's own brand of machine language; the translated, machine language version, or object program, can then be executed directly by the machine. The task of translating a program from Pascal to machine language is performed by the Pascal compiler, a system program that is available on command to any user of the system. A typical computer system includes a number of compilers, each for a different language--Pascal, Fortran, Basic, and so forth. When we submit a program for execution, we must tell the system that we wish to use the Pascal compiler to compile and execute a program. Thus, assuming that punched cards are the standard input medium on our system, we must submit for execution a complete job deck that includes our program, any data it is to read while executing, and some job control information to tell the system that the actions we want performed on this run are the compilation and execution of a Pascal program. The details of job deck setups and job control languages vary widely from one computer system to another, and are not part of the Pascal language itself, so we cannot say anything specific about them here. Your instructor will supply you with all you need to know to run Pascal programs at your local installation. The example that follows is intended only to give a very general idea of what a complete job deck includes.

Program G1.2 shows a typical job deck for running a Pascal program under the Scope operating system for the CDC 6000-series computers. The deck is divided into sections by separator cards, represented here by lines containing nothing but a percent sign (%) in column 1. (Actually, the standard form of separator in the Scope system is a card with a 7, 8, and 9 multiple-punched in column 1.) The first card in the deck is a user identification card, or job card, that identifies the user by a system account number (J9999) and specifies various parameters for the job (priority level, time limit in seconds, memory requirements, expected number of output pages). This is followed by a series of job commands, or control cards, terminated by a separator card. Each control card denotes a specific action, or jobstep, that the system is to perform. In this case two such actions are requested. The card that reads

```
PASCAL.
```

invokes the Pascal compiler to translate a program from Pascal into machine language; the card reading

```
LGO.
```

calls for the resulting object program to be loaded into memory and executed. (LGO stands for Load and GO.)

Program G1.2 Example job deck

```
J9999,7,1,50000,1.  SMITH      MAGIC NUMBER PROGRAM
PASCAL.
LGO.
%
PROGRAM MAGIC (INPUT, OUTPUT);

  (* EXAMPLE PASCAL PROGRAM TO GENERATE A ROW OF SOMEONE'S FAVORITE
     NUMBER, USING THE COMPUTATION DESCRIBED IN THE TEXT. *)

  VAR A, B, C, D:  INTEGER;

  BEGIN (* MAGIC *)
    READLN(A);
    B := 3 * A;
    C := 3 * B;
    D := C * 12345679;
    WRITELN('YOUR FAVORITE NUMBER WAS ', A);
    WRITELN('THE RESULT IS ', D)
    END (* MAGIC *).
%
4
%%
```

- -

The remaining sections of the job deck contain the input to be supplied to the various jobsteps, arranged in the order in which it will be needed. Since the first jobstep in this case is to activate the Pascal compiler, which expects a Pascal program as input, we place our program Magic immediately after the control cards in the job deck, terminated by another separator card. (The Pascal program in Program G1.2 is shown entirely in uppercase, because the CDC Scope operating system does not recognize lowercase letters. Some Scope systems also do not recognize the single quote character ' used in a few places in Program G1.2, so some other character, such as ≠ or ≡, must be substituted. This is an example of the kind of implementation-dependent detail that you will have to find out for your own local Pascal installation.) The next jobstep is the execution of the object program, which expects one line of input containing our "favorite number." Accordingly, the next section of the job deck consists of a single card of execution input containing the number 4. In the present example there are no further jobsteps, so the execution input is followed by a terminator card marking the end of the entire job deck. (The terminator card is represented here by percent signs in columns 1 and 2; on an actual Scope system it would be multiple-punched with a 6-7-8-9 in column 1.)

The printed output from a job is also normally arranged by jobstep. A typical printout from a Pascal run might include the following:

1. Preliminaries. A cover sheet of some sort containing the user's name or identification number in bold block letters for easy visual identification. This is often followed by a page of current news and bulletins of interest to users of the system.
2. Compilation listing. A copy of the program as it was read in by the compiler. There may also be some compilation statistics, such as the amount of CPU time and memory space used in compiling the program.
3. Load map. Produced on many systems whenever a user program is loaded into memory. It contains information that may be useful in locating the source of program errors, but can generally be ignored by the novice programmer.
4. Execution output. Any output produced by the Pascal program itself when it is executed. If all goes well, our execution output from Program G1.2 should consist of two lines:

```
Your favorite number was 4
The result is 444444444
```

5. Job Summary. A job log, or dayfile, containing information and statistics summarizing the job just run. Most of this information is of no immediate interest to the beginner.

We have been very careful to say, under point 4 above, that the execution output should appear as shown "if all goes well." Errors are a regrettable fact of life for all programmers, from the beginner to the expert. Very few programs run without a hitch the first time they are tried. Because of this fact, most computer systems provide diagnostic facilities to help detect errors when they occur, identify their causes, and correct them. Thus the compilation listing may include diagnostic messages to call our attention to errors and illegalities detected in our program by the Pascal compiler. Some of these may be mere warnings that the program seems to be doing something slightly unusual; others represent errors so serious that the compiler cannot produce a usable object program, and must suppress the loading and execution stages of the job. In such instances, of course, the output normally produced by those jobsteps will not appear. At other times the program may compile successfully, but an error may arise during the execution phase. When this happens, the execution output may be replaced or interrupted by some form of diagnostic information, such as a control trace or a memory dump, to help pinpoint the source of the problem. At first you will probably have to consult an expert to help you decipher these, but eventually you will want to learn how to interpret them for yourself. The use of such diagnostic aids to isolate and correct program errors is one of the skills that every programmer must acquire.

PROBLEM

1. Learn the rules for submitting a simple Pascal job at your local computer installation, and use them to prepare and run a copy of Program G1.1.

Our purpose in this chapter is simply to look at an example of what a Pascal program is like. We are concerned here only with the overall structure of the program, not with the details of the various statements it contains. These will all be taken up in the appropriate chapters of Part P.

Pascal is very similar in spirit and general structure to the informal "pseudo-language" used in the main text (which, since it has no name of its own, will be known throughout this book as "the text language" or "the language of the text"). Many of the differences between the text language and Pascal are merely superficial variations of essentially the same program structures, so that some statements can be converted from one language to the other in a purely mechanical way. However, as we will discover later on, there are also more significant differences between the two languages.

Program G2.1 is a Pascal version of the simple income-tax program displayed in the main text as Program G2.1a. In addition to comments that have been inserted to help explain the program to the human reader, the program consists of three main parts: a heading, a declaration part, and a body. The heading, always the first line of the program, announces the name of the program (in the present case, ComputeTax) and lists its external parameters (INPUT and OUTPUT). The meaning and purpose of these will not become clear until Section P11.3: until then, all you need to know is that in most programs the names INPUT and OUTPUT must be included as shown in the program heading.

- -

Program G2.1 Income tax calculation

```
program ComputeTax (INPUT, OUTPUT);

  (* Program to compute an individual's tax liability under the
     simplified rules given in the text. *)

  var Income, Dependents, Expenses, Deduction, TaxableIncome, Tax: REAL;

  begin (* ComputeTax *)
    while TRUE do (* Compute tax for one individual *)
      begin
        (* Read data *)
        READLN(Income, Dependents, Expenses);

        (* Compute maximum deduction *)
        Deduction := 0.15 * Income;
        if Deduction < Expenses then Deduction := Expenses;

        (* Compute taxable income *)
        TaxableIncome := Income - Deduction - (750.00 * Dependents);

        (* Compute tax due *)
        if TaxableIncome < 4000.00
          then Tax := 0.15 * TaxableIncome
          else Tax := 600.00 + 0.2 * (TaxableIncome - 4000.00);
        if Tax < 0.00 then Tax := 0.00;

        (* Print results *)
        WRITELN('Tax liability is $', Tax:0:2)

      end
  end (* ComputeTax *).
```

Following the heading is a brief explanatory comment. Anything written in a Pascal program between the special brackets (* and *) (or the "curly braces" { and } on systems where these characters are available) is assumed to be a comment and is simply ignored by the Pascal compiler. Thus, as far as the computer is concerned, comments have no effect whatever on the meaning of the program, and could be removed without changing the program's behavior in any way. Still, comments are important, because they help make the meaning of the program as clear to the people who read it as it is to the computer. A good habit to develop is to begin every program you write with an introductory comment explaining the purpose and method of the program to the human reader.

The next significant part of the program after the heading is the declaration part, which in Program G2.1 consists of the single line

```
var Income, Dependents, Expenses, Deduction, TaxableIncome, Tax: REAL;
```

The purpose of the declaration part is to define, or declare, the meaning of every name used in the program. Such names are called identifiers, and must conform to certain rules, which will be spelled out in Chapter P1. Notice that Standard Pascal allows the use of lower- as well as uppercase letters in an identifier. We will make use of this privilege to establish typographical conventions, described below, to distinguish the various kinds of identifier that can occur in a Pascal program. You must understand, however, that these are only conventions that we have adopted for use in this book, and that they are not meaningful, and may not even be possible, in your own local version of Pascal. (Many computers cannot handle lowercase letters. You will have to find out from your instructor about the use of upper- and lowercase in your local Pascal system.)

Certain identifiers, called reserved words, have special meanings that are built into the Pascal system. These are of two kinds, keywords and standard identifiers. Keywords are words that are integral to the language itself, and serve to denote the structure of the program: they are written throughout this book in underlined lowercase. Keywords appearing in Program G2.1 are program, var, begin, end, while, do, if, then, and else--we will learn all of their meanings in due course. Standard identifiers, which we will write in uppercase, are predefined by the Pascal system as the names of "built-in" constants, data types, procedures, and functions provided by the language. Program G2.1 includes the standard identifiers INPUT, OUTPUT, REAL, TRUE, READLN, and WRITELN. A complete list of Pascal keywords and standard identifiers is given in Appendix C3 of this manual.

Apart from reserved words, all identifiers used in a program must be defined by the programmer in the declaration part. If a program refers to an identifier that is not declared and is not recognizable as a keyword or a standard identifier, it is an error and the program will not be run. In this book user-defined identifiers are written in lowercase with an initial capital. (We will also capitalize each new word in the middle of an identifier, as in TaxableIncome in Program G2.1, since the underline character used in the text to separate words within an identifier is not available in Pascal.)

The declaration part of a Pascal program may include variable declarations (discussed in Section P1.1), label declarations (P3.1), type declarations (P4.1), constant declarations (P9.5), and subprogram (procedure and function) declarations (Chapter P7). Those declarations that are present in a given program must appear in the following order:

1. Label declarations
2. Constant declarations
3. Type declarations
4. Variable declarations
5. Procedure and function declarations (intermixed in any order)

All the declarations of each kind must be grouped together, preceded by an identifying keyword: label, const, type, or var. (Procedure and function declarations are treated somewhat differently--see Chapter P7 for details.) The identifying keyword is not included in every declaration, but appears only once for the entire group of declarations of a given kind. For example:

```
label 10, 15, 20, 100;
const NameSize = 20;
  ListSize = 50;
type Name = packed array [1..NameSize] of CHAR;
  ListIndex = 1..ListSize;
  NameList = array [ListIndex] of Name;
var Students: NameList;
  N1, N2: Name;
  I, J: ListIndex;
  X, Y, Z: REAL
```

The declaration part is followed by the body of the program, which in Program G2.1 consists of everything from the line

```
begin (* ComputeTax *)
```

to the line

```
end (* ComputeTax *).
```

The program name, ComputeTax, has been inserted here strictly as a comment, not as part of the program proper. The name of the program is not repeated on the last line in Pascal, as it is in the language of the text. However, the period on the last line is required, to mark the end of the program. Every complete Pascal program must end with a period.

The keywords begin and end are used in Pascal as statement brackets, to enclose a sequence of statements that are to be treated as a single unit. Any sequence of statements enclosed between begin and end is called a compound statement, and represents a single action to be performed, though we can make this single action arbitrarily complicated. A compound statement can be used wherever the rules of the language call for a single statement. Notice that the statement brackets begin and end always travel in matched pairs, like left and right parentheses. Like parentheses, they may be nested to any depth, as illustrated in Figure G2.1, where we have included identifying code letters as comments after each begin and end to show which pairs match.

- -

Figure G2.1 Nested compound statements

```
begin (* A *)
  . . . ;
  begin (* B *)
    . . .
    end (* B *);
  . . . ;
  begin (* C *)
    . . . ;
    begin (* D *)
      . . .
      end (* D *);
    . . .
    end (* C *);
  . . .
  end (* A *)
```

The body of a program is always a compound statement: even if only one statement is needed to describe what the program does, that one statement must be enclosed in statement brackets to turn it into a compound statement. This is the case, in fact, in Program G2.1, whose body contains only one statement, of the form

```
while TRUE do
  begin
    . . .
  end
```

(This is Pascal's equivalent of the do forever loop used in the corresponding program of the main text.) As we will learn in Chapter P3, the body of a while loop is always a single statement. Since, in this case, the action to be taken on each pass through the loop requires seven statements to express, we place them between begin and end to turn them into a single compound statement. Notice that this gives us an inner compound statement, representing the body of the while loop, nested within an outer compound statement that forms the body of the entire program.

The statements in the body of the while loop in Program G2.1 correspond fairly closely to those in the original text version, Program G2.1a. The names of the substeps used in Chapter G2, where this program was developed by stepwise refinement, have been retained as comments in the Pascal version, to help guide the reader through the program. Each such comment is followed by one or more statements representing the realization of that substep in Pascal code. This kind of documentation is an important habit to acquire in writing your own programs, and is a natural by-product of the technique of programming by stepwise refinement discussed in Chapter G2.

A comparison of Programs G2.1 and G2.1a reveals only a few superficial differences in form between Pascal and the text language. These will be discussed fully in later chapters. Notice that the Pascal standard identifiers READLN and WRITELN (READ a LiNe and WRITE a LiNe) correspond to the keywords input and output in the text. One significant difference between the two programs results from the fact that Pascal has no built-in operation like MAX in the text, for finding the larger of two numbers. Hence we must do the equivalent operation "by hand," replacing the statement

```
DEDUCTION ← MAX(EXPENSES, 0.15 * INCOME)
```

in the text with the pair of Pascal statements

```
Deduction := 0.15 * Income;
if Deduction < Expenses then Deduction := Expenses
```

Similarly, the statement

```
TAX ← MAX(TAX, 0)
```

in the text version is replaced with

```
if Tax < 0.00 then Tax := 0.00
```

in Pascal.

Looking closely at Program G2.1, you will observe that some lines end with a semicolon and others do not. A semicolon is not part of the statement it follows. Rather, it is used as a separator between two statements or other program elements that are to be executed or interpreted consecutively, one after the other. Semicolons are needed to separate

- The program heading from the declaration part
- Consecutive declarations within the declaration part
- The last declaration in the declaration part from the program body
- Consecutive statements within a compound statement

Notice, in Program G2.1, that the conditional statement

```
if TaxableIncome < 4000.00
  then Tax := 0.15 * TaxableIncome
  else Tax := 600.00 + 0.2 * (TaxableIncome - 4000.00)
```

is followed by a semicolon, to separate this statement from the one after it, but that the if, then, and else clauses within the conditional statement are not separated by semicolons, since they are not separate statements to be executed consecutively. Similarly, there is no semicolon after the line

```
while TRUE do
```

because what follows is the body of the same while statement, not another statement to be executed next. Notice also that there is no semicolon on the statement

```
WRITELN('Tax liability is $', Tax:0:2)
```

Here the reason is that this is the last statement in the compound statement forming the body of the while loop: the keyword end on the following line is not a statement to be separated from the line before, but a punctuation mark--a statement bracket--marking the end of the compound statement. Semicolons separate the statements in a compound statement just as commas separate the elements of a parenthesized list. Inserting a semicolon before the terminating end of a compound would be like writing

```
(A, B, C,)
```

instead of

```
(A, B, C)
```

P1 DATA

Every item of data that occurs in a Pascal program has a definite type, which determines the kind of information the item represents, the way it is represented in memory, and the operations that can be performed on it. The Pascal language provides facilities for manipulating many different types of data, from simple "built-in" types--integers, real numbers, text characters, and logical values (TRUE and FALSE)--to more complicated structures such as arrays, records, files, and pointers. We will introduce all of these types in the chapters to come; for the moment, we will restrict our attention to the basic numeric data types INTEGER and REAL.

There are two different ways in which a data item can be represented in a Pascal program: as a constant or as a variable. A constant "stands for itself"--that is, it is written in a form that represents the data value directly, such as -13 or 4.37. The type of the constant is determined by the form in which it is written: the constant -13 will be recognized as an integer and 4.37 as a real number. The specific rules for writing constants in Pascal will be given later in this chapter. A variable is referred to by writing the name, or identifier, given to it by the programmer; its type must also be specified by the programmer, as we will see in the next section.

P1.1 Variable Types

Every variable used in a Pascal program must be declared by the programmer, either at the head of the program or at the head of the subprogram (procedure or function) in which the variable is used. Since we will not encounter subprograms until Chapter P7, we will assume for the moment that all declarations appear at the head of the program itself.

A Pascal variable declaration consists of the name (which must conform to the Pascal rules for identifiers, given below) and the type of the variable, separated by a colon. For example,

```
Count: INTEGER
```

declares a variable named Count, of type INTEGER. Two or more variables of the same type may be included in a single declaration, such as

```
Velocity, Distance, Time: REAL
```

which declares three real variables with the names given. In a Pascal program, all the variable declarations must be grouped together in a variable declaration part, separated by semicolons, with the entire list preceded by the keyword var. For example, the variable declaration part for the variables declared above would be

```
var Count: INTEGER;
  Velocity, Distance, Time: REAL
```

Notice that although there may be any number of variable declarations, the keyword var appears only once, at the beginning of the list.

The rules for forming identifiers in Pascal are essentially the same as those given in Chapter P1 and used throughout the main text. An identifier is a sequence of alphanumeric characters (letters and digits), the first of which must be a letter. There must be no "embedded blanks": an identifier such as

```
Taxable Income
```

is not allowed. The only differences between Pascal identifiers and those used in the language of the text are:

- As noted in Chapter G2, the use of the underline character as a placeholder to separate words within an identifier is not permitted in Pascal. Thus identifiers such as NEXT_ENTRY, used in the main text, are not allowed.

- Standard Pascal recognizes both upper- and lowercase letters. This is not true of all implementations of the language, however, since many computers and input/output devices cannot handle lowercase letters.
- Standard Pascal allows identifiers of any length, but in some implementations only the first eight characters are significant: that is, two identifiers that begin with the same eight characters are considered identical. Thus the names Variable1 and Variable2 refer to the same variable. To avoid unpleasant surprises, it is a good idea on such systems to limit all identifiers to eight characters or fewer. Some Pascal implementations may place even stricter limits on the length of identifiers.

Not all identifiers used in a Pascal program represent variables. As we will see in later chapters, identifiers may also be declared to stand for constants, types, procedures, and functions. In addition, some identifiers are treated as reserved words that play a special role in the Pascal language. As we have seen in Chapter G2, such reserved words are of two kinds. Keywords, such as program, if, and var, serve to define the structure of the program itself, and may not be used as programmer-defined identifiers: a declaration such as

```
var if: INTEGER
```

is an error. Standard identifiers are the names of built-in constants, types, procedures, and functions provided by the Pascal language (FALSE, INTEGER, SQRT), and may be "redeclared" by the programmer. This means that a declaration like

```
var SQRT: REAL
```

is perfectly legal: the identifier SQRT is redefined as the name of a newly created real variable, losing its standard meaning as the name of the built-in square-root function. Any program containing such a declaration would thus be unable to refer to the standard

---

Program P1.1 Average of a set of numbers

```
program FindAvg (INPUT, OUTPUT);

  (* Read a set of nonnegative real values and calculate their aver-
     age.  First negative value encountered signals end of input. *)

  var Average, Total, Value: REAL;
    N: INTEGER;

  begin (* FindAvg *)

    (* Initialize variables and read first input value *)
    N := 0;
    Total := 0.0;
    READLN(Value);

    (* Count and total values until a negative value is encountered *)
    while Value >= 0.0 do
      begin N := N + 1;
        Total := Total + Value;
        READLN(Value)
        end;

    (* Compute average and output results *)
    Average := Total / N;
    WRITELN(N, Average)

    end (* FindAvg *).
```

---

square-root function, since that function would no longer have a name! For this reason, it is wisest to avoid using standard identifiers as variable names (or as names of programmer-defined constants, types, procedures, or functions). A complete list of Pascal keywords and standard identifiers is given in Appendix C3.

Program P1.1 is a Pascal version of Program P1.1 in the text. Of the four variables declared in the program, three (Average, Total, Value) are of type REAL, and one (N) of type INTEGER. The standard input/output procedures READLN and WRITELN can tell from the declarations what form of data representation to use; similarly, the Pascal compiler can tell what form of arithmetic (integer or floating-point) to use for a statement like

```
Total := Total + Value
```

by examining the declared types of the variables used.

P1.2 Constant Types

In addition to the variables Average, Total, Value, and N, Program P1.1 refers to three constants: the integer constants 0 and 1 and the real constant 0.0. An integer constant is simply a string of one or more decimal digits, optionally preceded by a sign (+ or -). Thus

```
8352     +4     -209
```

are all valid integer constants. Notice that no blanks are allowed between the sign and the number.

As in the case of identifiers, the rules for writing real constants in Pascal are basically the same as those given in Chapter P1 of the text. To be recognized as a real constant and not an integer, a number must contain a decimal point, a decimal exponent preceded by the letter E, or both. The exponent and the number as a whole may each have an optional sign; these signs are independent and need not agree. The following are all valid real constants in Pascal:

```
1.732          -273.18       6.02E+23      -8.317E3
6.625E-34      -1.6E-19      3E8           +2E-11
```

Notice again that blanks may not be embedded anywhere within a constant. The only difference between Pascal's real constants and those in the text is that, if a decimal point is present, Pascal requires at least one digit on either side of the point. Thus the constants

```
4096.          .707
```

which would be considered valid in the text language, must be written

```
4096.0         0.707
```

in Pascal. Similarly, if a real constant contains an exponent, there must be at least one digit before the exponent. Thus

```
E-6
```

is not a valid Pascal constant, and must instead be written

```
1E-6
```

The range of numeric values that can be represented and manipulated in a Pascal program depends on the word size of the host computer. On the CDC 6600, with its spacious 60-bit memory word, we can represent any integer in the range ±281474976710655 (2 to the 48th power - 1). Real values may have magnitudes as large as 10 to the 322nd or as small as 10 to the -322nd power, with up to 48 bits (about 15 decimal digits) of precision. On a machine with a more modest word size of 32 bits, integers might run between ±2147483647 (2 to the 31st power - 1), and reals between 10 to the 38th and 10