A CS-4 PRIMER

VOLUME I

BASIC FEATURES

Second Edition

February 1975

Robert Fourer

James L. Felty

Intermetrics, Inc.
701 Concord Avenue
Cambridge, Massachusetts 02138

Prepared for: Contract N00123-74-C-00634
Naval Electronics Laboratory Center
San Diego, California 92152

Sponsored by: Naval Air Systems Command
Naval Electronic Systems Command
Washington, D. C. 20360

This document is Data Item CDRL A002 A CS-4 Primer: Volume 1: Basic Features, under Contract N00123-74-C-0634. Mr. Warren E. Loper, Code 5200, is the Contract Technical Monitor for the Navy. Dr. James S. Miller is Project Manager for Intermetrics.

PREFACE

This is the second edition of A CS-4 Primer: Volume 1: Basic Features. It reflects the revisions made in CS-4 through January, 1975, and thus supercedes the previous draft publication of Volume I (January, 1974). This volume presents the basic features of CS-4; it is to be followed by subsequent volumes which whill cover all of the more advanced features of the language.

Design criteria

The Primer may be used to teach the CS-4 language to programmers of widely differing experience and capability. It may be employed in varying educational contexts. To ensure its usefulness in a variety of situations, the following guidelines have been followed in the design and writing:

1) The Primer should assume no prior knowledge of computers or computer languages, so that it can be used to teach CS-4 as a first language.

2) The Primer should also cover the more advanced aspects of CS-4 which will be taught to experienced programmers.

3) The Primer should be structured so that readers with programming experience can easily skip over the more elementary parts of the exposition.

4) The Primer should not assume any degree of supplementary instruction. It should be usable with or without classroom lectures, assigned machine problems, and the like. To this end, all points of importance should be introduced with numerous examples which show the reader how to make use of what he has learned.

5) Good programming practices should be stressed, both by the general approach taken, and in the specific examples.

6) The implementation-dependent aspects of programming should be left out or played down as much as possible.

7) The prose style should be simple and conversational, but not condescending.

TABLE OF CONTENTS

PART 1

ELEMENTARY STATEMENTS AND

DATA TYPES

1.0

INTRODUCTION: COMPUTER PROGRAMS AND PROGRAMMING LANGUAGES

How to use this Primer

This Primer is an introduction to writing computer programs in the programming language CS-4. If you are unfamiliar with writing programs for a computer, you should keep reading right here. If you have had some experience with a programming language, you may prefer to skim the introductory chapters, and concentrate on those which treat concepts that seem unfamiliar. If you have already done a substantial amount of programming, you may want to skip directly to Volume 2 of the Primer, which contains a summary of all aspects of CS-4 which are introduced in this volume.

However you read this Primer, it will be of most use to you if you write and run your own programs while you are going through it. The Primer itself contains some sample programs, which you might also like to run, perhaps with your own modifications. Actually writing and correcting programs is the best way to learn how CS-4 is used.

Once you get past the most elementary chapters, you will probably want to keep handy copies of the two reference manuals for the language -- the CS-4 Language Reference Manual and CS-4 Operating System Interface. They contain a concise and complete description of all features of CS-4, including a number of special and advanced points which, for simplicity, are omitted from this Primer.

What is a computer program?

In order to do something with a computer, you have to write a program for it. A program is a set of instructions that tells the computer how to do what you have in mind. Without these written instructions -- telling it exactly how to do the job -- a computer cannot perform any task for you at all.

You may not be familiar with this meaning of the word "program". It has been used this way only since the invention of the computer. But the idea behind it -- a set of precise instructions to be carried out

literally and in some order -- is one you have encountered many times. A carefully marked road map is one obvious example -- it is a way of listing, in order, the various left and right turns a driver must make to reach a certain destination. Another example is a recipe, whose detailed instructions enable a cook to produce an unfamiliar dish. A program is also a series of instructions, but one that can be carried out by a computer.

A computer does not carry out instructions in quite the same way that a person does. For one thing, it works considerably faster than a person ever could. It also has a much greater capacity for storing and retrieving detailed data. A third difference is perhaps less obvious: a computer follows its instructions literally, much more so than, say, human beings follow a recipe. If a recipe calls for 11 cups of sugar, you may recognize it as a misprint and use 1 cup instead. But if an instruction you give the computer can be carried out, it will be carried out regardless of whether or not it makes sense in the context of what you are trying to do. Moreover, if an instruction cannot be carried out -- because it is misspelled, or otherwise impossible to interpret as stated -- the computer will cut its work short and report that it has encountered an error. A friend or a cook might hazard a guess at what you meant by an incorrect instruction, but the computer will not. So a computer program must be written with special care, to be sure that it says exactly what you want it to, and to be sure that it says it in the right way.

What is a programming language?

A computer must be given instructions in a language it can understand. Such a language is called a programming language. You might wish to program a computer in English, since that is a language you use to communicate with other people as well. But computers cannot understand English; they can be programmed to recognize a few dozen, or even a few hundred, English words, but an entire natural language is too complex and too ambiguous for them. What is more, English is not well suited to describing many of the things you might want a computer to do. Numerical calculations, for example, are more easily and more concisely described using the symbols of arithmetic. So programming languages usually describe calculations with algebraic notation, instead of with words.

Programming languages also incorporate the formulas of mathematical logic, because they are an especially brief way of stating conditions that can be true or false.

The traditional notations of arithmetic, algebra, and logic were not devised with the computer in mind. Hence they are not sufficient to describe everything you might want a computer to do. What is needed is a language more extensive than mathematical notation, but more restricted and more precise than natural English. A language that meets these needs will be suitable for programming a computer.

This Primer introduces the programming language CS-4. A set of instructions written in CS-4 can also be carried out by any person who understands the language: they do not have to be done by a machine. But the important thing is that a computer can understand them, and can carry them out much faster and more reliably than a person could.

How does the computer understand the programming language?

Each type of computer is built to understand one particular language -- its own machine language, so to speak. This language is designed to be stored in the computer as patterns of electronic signals, and it is interpreted automatically by the computer's circuitry. It is possible to write programs in machine language, by encoding the proper electronic patterns on a tape or some other medium that can be read by the computer; but such a process is extremely cumbersome and time-consuming. Moreover, a program in one computer's machine language will not be understood by other computers.

Fortunately, you do not have to know anything about machine languages in order to use a computer. You can write all of your programs in CS-4. The computer will translate your CS-4 instructions into its own internal machine language instructions. Then it will carry out the translated instructions.

The computer translates CS-4 instructions into its internal language by following a special set of instructions called the compiler. This process of translation is thus known as compiling. Your entire program -- all of your instructions for doing a particular thing -- is compiled all at once. After the translation is complete, the computer begins to carry out the instructions in the proper order. When the machine

is actually carrying out the instructions we say it is executing the program.

So you see there are two steps to running any CS-4 program on the computer: compilation and execution. If the compiler finds an instruction in your program that it cannot understand, or that is written improperly, it will not be able to translate that instruction. As a result, it will not allow the program to be executed. You will have to correct the erroneous instruction and submit the program to the compiler again.

Once the compiler accepts a whole program, it can be executed; however, that does not guarantee that it is error-free. The compiler can only tell that the program is written properly, according to the rules of the language. It cannot tell whether it is written properly for the computer to do what you wanted it to do. You have to check for that yourself.

Practically every program (unless it is extremely small and simple) initially has a couple of errors or "bugs", as they are commonly known. The process of correcting all the errors -- called debugging -- is every bit as important as writing the program in the first place. Often a careless programmer must spend more time debugging his work than writing it.

Writing readable programs

In CS-4, as in most languages, there is usually more than one way to give the same set of instructions. To the computer, it usually matters little which way the instructions are given -- it follows them with its customary speed and precision in any case.

It is important to keep in mind, however, that although your programs are always executed by a computer, they will have to be read by people, too. You will have to read them yourself -- when you are debugging, or when you want to make changes to a program after it is running properly. Often other people will want to read your programs, too -- to modify them for their own use, or to look for errors you might not have caught. Even when you are first writing a program, it is valuable to organize and write your instructions in a way that makes it easy for you to keep track of what you are doing.

One of the objectives of this Primer is to help you develop a good CS-4 "style". To do this, we have tried to present many sample programs which are relatively easy to read and understand. The text also makes some explicit comments about readability and good programming practice.

2.0

ARITHMETIC AND STORAGE

People usually think of a computer as a machine that works with numbers. It can add, subtract, multiply, and divide them. It can put a number in storage, and recall it later.

A useful general programming language must be able to tell the computer how to do arithmetic and storage. This chapter explains how you write these operations in CS-4.

Numerical values

You can write numbers in CS-4 the same way you do in arithmetic. A decimal point is allowed, but it is not required. For example:

```
12
32.731
7
7.
7.0
.8
0.8
0.0001728901
627000000000000
```

Commas or spaces within numbers are not accepted by the compiler. For example, if the last example above were written

```
627,000,000,000,000  or  627 000 000 000 000
```

it would not be correctly recognized as a number, and would probably be rejected outright as an error (whether the compiler can detect the error will depend on the context in which the error appears).

Numbers written in a program are called literal constants (or simply literals), since their values are indicated by the way they are written.

Literals with exponents

There is another type of numerical literal in CS-4 which might not be as familiar to you. It is a literal with an exponent part -- an adaptation of the "scientific notation" used in chemistry and physics. It consists of a mantissa value, followed by a letter E, followed by an integral exponent value; it represents the mantissa multiplied by 10 to the power of the exponent. Here are some examples:

With exponent	Equivalent without exponent
2.3E4	23000.
2.3E-4	.00023
2.3E00	2.3
147.E3	147000
.091E-5	.00000091
2E+25	20000000000000000000000000
7.9375892E-15	.000000000000079375892

The last two examples demonstrate how literals with exponents conveniently represent very large or very small magnitudes.

The exponent part of the literal must not have a decimal point; it may be preceded by a plus or minus sign, as the examples show. Spaces or commas within the literal are prohibited, as they are in non-exponential literals.

Arithmetic operations

You can write addition, subtraction, multiplication, and division in CS-4 in much the same way as they are usually written in arithmetic:

2 + 3	2 plus 3
2 - 3	2 minus 3
2 * 3	2 times (multiplied by) 3
2 / 3	2 divided by 3

(* has been chosen for multiplication, and / for division, because they are much more commonly found on card punches and printers than the traditional × and ÷).

You can also use a minus or plus sign in front of a single operand. The former negates a value, while the latter leaves it the same:

```
-2.3E-4          the negative of 2.3E-4 (-.00023)
12 * -6.7        the product of 12 and negative 6.7 (-80.4)
2 ** -4          2 to the power -4 (0.0625)
2 ** +4          2 to the 4th power (16)
```

CS-4 does not recognize the usual superscript notation for raising a number to a power (2^5, for instance, for 2 to the 5th power). Instead, there is a symbol for exponentiation. It is written as two * characters, with no space between them:

```
2 ** 5           2 to the 5th power
2 ** 0.5         2 to the power 0.5 -- the square root of 2
```

Operators

Symbols such as +, -, *, **, and / are referred to as operators. The entities (such as numbers) that they operate upon are called their operands. When operators are used in expressions like these:

```
4 + 5
3.2 - 15
12 * 700
2 ** 15
0.05 / 0.03
```

they are called infix operators, because they are placed in between their operands. When a plus or minus sign is used like this:

```
0.5 * -6.3
-3 / +4
```

it is called a prefix operator, because it appears before its one operand. CS-4 has other infix and prefix operators, which you will learn about in the next several chapters.

In general, the amount of spacing between operator and operand(s) makes no difference. For example, the following all are legal, and mean the same thing:

```
2 + 3
2+3
2 +3
2+ 3
2  + 3
```

Programs are usually most readable, however, if the first example above is followed.

Many operators of CS-4 are written, like **, with two or more characters. Do not leave any spaces between the characters of a multi-character operator: the characters must be adjacent, or the compiler will not interpret the operator as you intended. For instance,

```
2 * * 3
```

will be read as having two * operators between the literals, instead of one ** operator.

When two operators appear in succession, they must be separated by at least one space. For example, you can write "six times negative three" in any of these ways:

```
6 * -3
6* -3
6 * - 3
```

but if you write it like this:

```
6*-3
```

your intentions will not be understood by the compiler. The compiler will not be able to tell that * and - are supposed to be separate operators. (You can play it safe by leaving spaces on both sides of your operators; that way, the compiler will always read them as you intended.)

Precedence and associativity

When more than one operator is used in an expression, the compiler must have some way of determining just what each operator's operands are. For instance, in

```
7.3 + 6.9 * 2.0
```

is 6.9 the right operand of +? Or is its right operand 6.9 * 2.0? One way to distinguish alternatives such as these is to group operands in parentheses, just like it is done in algebra:

`7.3 + (6.9 * 2.0)`	7.3 plus the product of 6.9 and 2.0 (21.1)
`(7.3 + 6.9) * 2.0`	2.0 times the sum of 7.3 and 6.9 (28.4)
`-(5.5 - 2.1)`	the negation of 5.5 minus 2.1 (-3.4)
`(-5.5) - 2.1`	negative 5.5 minus 2.1 (-7.6)

If the grouping of operations is not fully indicated by parentheses, it is determined by the precedence of the operators. Operators with higher precedence are applied (grouped with their operands) before those of lower precedence. The operators we have introduced so far are ranked as follows:

`infix **`	highest precedence
`prefix + -`	
`infix * /`	
`infix + -`	lowest precedence

So ** is applied before the prefix arithmetic operations, which are applied before * and /, which are applied before infix + and -. For example:

Expression	Equivalent expression with parentheses
`7.3 + 6.9 * 2.0`	`7.3 + (6.9 * 2.0)`
`-5.5 - 2.1`	`(-5.5) - 2.1`
`-2 ** -5 * 3 - 10`	`((-(2 ** (-5))) * 3) - 10`

Sometimes two infix operators with the same precedence occur together, in expressions like:

```
2 / 8 * 4
3.4 - 2.1 + 0.9
2 ** 3 ** 0.5
```

If the operators are +, -, *, or /, the leftmost one is applied first; we say that these operators are left associative. On the other hand, ** is right associative -- the rightmost occurrence of the operator is applied first. Thus the examples above are evaluated like this:

Expression	Equivalent expression with parentheses
2 / 8 * 4	(2 / 8) * 4
3.4 - 2.1 + 0.9	(3.4 - 2.1) + 0.9
2 ** 3 ** 0.5	2 ** (3 ** 0.5)

Storing a value

You can save a numerical value, or the result of a calculation, by assigning it a space in storage. A stored item of data is called a variable. Every variable has a name, which is associated by the computer with the value assigned to it.

To assign a value to a variable, you write an assignment, which looks like this:

```
DISTANCE := 12
```

The colon followed by an equals sign is called an assignment operator. An assignment causes the value expressed to the right of the assignment operator to be associated with the variable whose name is to the left of the assignment operator. In the example above, the value of 12 is associated with the variable named DISTANCE. You can read this as "12 is assigned to DISTANCE" or "DISTANCE gets the value 12" or just "DISTANCE gets 12".

Assignments often appear as individual statements in a program. A statement is a basic unit of a CS-4 program; in the chapters to come you will learn many types of statements in addition to assignment statements. A program's statements -- of any type -- are separated by semicolons. The compiler assumes that what follows a semicolon is intended to begin a new statement.

Here is a sample sequence of four assignment statements:

```
TEMP := 99.8;
DIFFERENCE := 9.7 - 9.6;
BAL := 50000 * (1 + 0.0015);
X := -1;
```

As you can see, the left operand of the assignment operator is always a variable name. It makes no sense to write

```
3 := 3.14159;
```

because the literal constant 3 has a fixed value; you cannot assign a new value to 3. The compiler will reject such an assignment as an error.

The spacing rules for := are the same as for any two-character infix operator. Hence, your program will not be interpreted by the compiler as you intended it if you write

```
X:=-1;              (space required between the two
                       operators := and -)
X : = 3 ;           (space not allowed between the two
                       characters of :=)
```

The semicolon is not an operator; the amount of space around it makes no difference.

(Spacing rules for CS-4 are quite liberal -- usually any reasonable spacing is allowed. The rest of this primer will therefore often omit detailed spacing rules. Instead, a preferred style will be shown in the examples. You can find complete rules for spacing in the Language Reference Manual.)

Using the stored value of a variable

Once a value is assigned to a variable, you can use the variable name to represent that value in an arithmetic expression. For example,

```
X := A + 2
```

takes the value associated with A and adds 2 to it; the result is then assigned to X. If you had previously written A := 3, then X would be assigned the value of 3+2.

Here are some more examples: assume you have made the assignments

```
DISTANCE := 30.;
TIME := 2.5;
WEIGHT := 1000.0;
```

Following are some subsequent assignments you might make, along with their meanings:

`RATE := DISTANCE / TIME;`	12.0 (30. divided by 2.5) is assigned to RATE
`WORK := WEIGHT * DISTANCE / 100;`	300 (30. times 1000.0 over 100) is assigned to WORK
`NEWRATE := (DISTANCE - 2) / TIME;`	9.2 (the quotient of 30. - 2 and TIME) is assigned to NEWRATE
`J := TIME / TIME;`	1 (2.5 divided by 2.5) is assigned to J

Storing a new value of a variable

Suppose you have already written

```
WIDTH := 5;
LENGTH := 3;
X := WIDTH * LENGTH;
```

which assigns the value 15 to X. Then later you write

```
X := WIDTH - LENGTH;
```

which assigns 2 to X. What happens to the 15? It is erased. When a new value is assigned to a variable, the old value is simply replaced; it is not saved. If you really wanted to save both WIDTH * LENGTH and WIDTH - LENGTH, you would have to assign them to different variables.

It is often very useful to be able to replace an old value of a variable with a new one, as you will see in the examples in the following chapters. Keep in mind, however, that assignments will be carried out in exactly the order you write them. For instance, if you write

```
A := 1;
B := 4;
C := 9;
B := A + 1;
C := B + 1;
```

the three assignments with only literals on the right will be performed first. Then the fourth will be executed, changing the value of B to 2. When the fifth assignment is executed, C's value is changed to 3 (2 + 1); the old value of B, 4, has already been erased.

Names for variables

The name of a variable must begin with a letter, which may be followed by up to 31 more letters or numerals. No spaces are allowed within names. However, the underscore character (_) is allowed within

names, and it is often used to give the appearance of a space. (Underscore may not be the first or last character of a name.) Characters other than letters, numerals, and underscore cannot be used in variable names.

You may often find it helpful to choose names that have some mnemonic significance -- TIME, DISTANCE, NEW_RATE, ACCELERATION_TIMES_10 -- but of course the compiler is not influenced by what a variable name may or may not mean in English. Nor can the compiler tell a right spelling from a wrong one. Every different spelling is treated as a different variable.

Names which meet the restrictions we have given are referred to as identifiers or words. Nearly any word may legally be used for a variable. However, there are certain ones, called reserved words, which have special meanings in CS-4. They cannot be used as variables -- any attempt to do so will result in an error message from the compiler. A full list of reserved words may be found in an appendix to the Language Reference Manual.

3.0

WRITING A SIMPLE PROGRAM

Once you know how to use variables and assignment statements, you are nearly ready to begin writing simple CS-4 programs which can be compiled and run on the computer. The other things you must know about to write a complete program are explained in this chapter.

Putting a program into a form the computer can read

Usually a program is first written by hand on paper. But in order to run it, you must transcribe it into a form that a computer can read. The two most common forms are:

1) Data cards punched on a standard keypunch.

2) Records which are entered on a disk or other storage device through a remote terminal.

The installation where you are working will teach you to use one of these methods before you are allowed to run programs. In this primer, for simplicity, we will assume that all programs are read from cards. However, you can substitute "record" for "card" throughout; the rules will be the same. In our sample programs, one line on the page will always represent the contents of a single card.

There is no required format for CS-4 statements. The compiler reads your program as if it were a continuous stream of characters, with the last column of each card directly followed by the first column of the next card. The end of a card never implies the end of a statement. Instead, statements are explicitly separated by semicolons.

As a result of this freedom, you can type statements anywhere on a card:

```
                    A := 1;
B := 2;
                                        C := 3;
```

You can type more than one statement on a single card:

```
A := 1;     B := 2;     C := 3;
```

And you can start a statement on one card and end it on a subsequent card:

```
A :=
1;          B :=
2;          C :=
3;
```

However, it is best to type simple statements like assignments one to a card and all in the same columns:

```
A := 1;
B := 2;
C := 3;
```

The statements are easiest to read this way, and it is easy to correct a single statement by just retyping it on a single new card.

Of course, it is important to keep your cards in the right order. In a simple program, the compiler assumes you want the statements executed in the order their cards were read in. (There are ways to have statements executed in different orders -- but that is a topic for subsequent chapters.)

Control cards

When you submit your punched program to be run, you will have to add a few special cards to it. These control cards will, among other things, give your name and account number, ask to use the CS-4 compiler, and tell where your program begins and ends.

The statements on control cards are not part of CS-4. They are part of a special language used by the computer installation on which you are running your program. Different installations require different control cards. You will be told what control cards to use before you are allowed to run programs.

Declaring variables

So far we have been assuming that any variable name can be used to store numerical values in any program. However, this is not really true. As you will learn, CS-4 variables can represent many classes (or kinds) of values besides numerical values (for instance, they may represent character strings, or arrays of one or more dimensions). Therefore it is necessary to declare in your program that your variables represent numerical values, so that statements which use the variables can be compiled correctly.

Each class of values which can be assigned to variables in CS-4 is called a mode. One mode, which contains numerical values, is named the REAL mode (the term "real" comes from mathematics). A variable may take values of only one mode; a variable of REAL mode, for example, can only represent numerical values.

To tell the compiler that your variables are of REAL mode, you write one or more declaration statements, like this:

```
VARIABLE HEIGHT IS REAL;
VARIABLES A1, A2, A3 ARE REAL;
```

The first word of the statement is always VARIABLE (or VARIABLES). It is followed by either a single variable name and IS REAL, or by a comma-separated list of variable names and ARE REAL.

Every variable in a program must appear in a declaration statement in the program. If an undeclared name appears -- either because of misspelling, or because its declaration was left out -- the compiler reports an error, and the program cannot be run.

Declaration statements differ in a basic way from other statements such as assignment. The latter are said to be executable -- because when the program is run they are executed, in a specified order, and each results in some specific action. Declarations are non-executable -- they are not part of the order of execution when the program is run. Instead, they provide a general sort of information to the compiler, such as the modes of variables; the compiler decides what actions are to be performed as a result of a declaration, and when they are to be performed. For instance, the above declaration of HEIGHT might cause storage for a numerical value to be set aside when the program begins.

In a simple program it is usually best to place declarations together at the beginning, before statements of any other type. Programs are more readable this way. Furthermore, placing declarations at the beginning guarantees they always have the same effect; in more complex programs, declarations not at the beginning may have somewhat different effects, depending on their context.

Limits on REAL values

CS-4 imposes no restrictions on the numerical values that may be represented by REAL variables. Unfortunately, there is no way a computer can actually store any arbitrary value in a fixed space. Thus, in practice, there are limits on the values that can be given to variables declared as REAL. These limits vary from machine to machine. But they are all of two basic types, which we can explain briefly here.

First, there are restrictions of magnitude. The absolute value of a REAL (its value with the sign made positive) may not exceed a certain limit; if a value greater than the maximum is generated by some calculation, an error condition called overflow occurs. An overflow normally results in an immediate end to execution of a program, and an error message in the printout. (There are ways to continue running after an overflow, but that is an advanced topic.)

The absolute value of a non-zero REAL must also be greater than a certain minimum. Otherwise, an error condition called underflow occurs. An underflow also causes execution to end.

The second type of restriction is one of precision. There is a limit to the number of significant digits that a REAL value can have. (The first non-zero digit and all subsequent digits are considered significant.) If no more than three significant digits could be stored, for instance, then both of:

```
HEIGHT := 3926;
HEIGHT := 0.0004070;
```

would be errors, since they assign HEIGHT values with a precision of 4 digits. For uniformity, all examples in this volume will assume 6 digits of precision.

Printing output

You can print out the values of variables, literals, or any other expressions by using the PRINT statement of CS-4. It is an executable statement, written like this:

```
PRINT (LENGTH, WIDTH, LENGTH * WIDTH, 0.0025);
```

The values associated with the items in parentheses are printed out in the order they are listed. If LENGTH and WIDTH above were declared as REALs, and had values of 1.5 and 37.5, respectively, then the following line would appear in the printout:

```
1.50000E+00 3.75000E+01 5.6200E+01 2.2500E-03
```

The items inside the parentheses of the PRINT statement must be separated by commas. There is no comma after the last item. If there is only one item, no commas are written:

```
PRINT (AREA);
```

There may be no items at all in the PRINT list:

```
PRINT;
```

in which case a blank line is left in the printout.

Comments

One way to make a program easier to read is to insert comments explaining what is going on. The start of a comment may be indicated by the symbol #, and its end by a second #. Alternatively, the start may be indicated by { and the end by }. The following are both valid comments:

```
# THE AREA IS COMPUTED #
{STEP #4:  THE OUTPUT VALUE(S) ARE PRINTED}
```

Any symbol may appear within a comment, except whichever end-comment symbol (# or }) is being used.

The content of a comment is never interpreted by the compiler. Instead, the entire comment is interpreted as if it were a space character. Hence a comment may be inserted anywhere a space is allowed.

A sample program

You now know enough to write an extremely elementary CS-4 program. As a sample of what you can do, we have chosen a simple commercial application: a billing program which updates accounts of the common revolving-credit type.

```
VARIABLES OLD_BAL, NEW_BAL, PURCHASE, PAST_DUE, INTEREST,
   PAYMENT ARE REAL;
OLD_BAL := 44.00;          # FIRST ACCOUNT #
PAYMENT := 44.00;
PURCHASE := 25.00;
PAST_DUE := OLD_BAL - PAYMENT;
INTEREST := PAST_DUE * 0.015;
NEW_BAL := PAST_DUE + INTEREST + PURCHASE;
PRINT (OLD_BAL, PAYMENT, PAST_DUE, INTEREST, PURCHASE,
   NEW_BAL);
OLD_BAL := 196.95;         # SECOND ACCOUNT #
PAYMENT := 25.00;
PURCHASE := 0;
PAST_DUE := OLD_BAL - PAYMENT;
INTEREST := PAST_DUE * 0.015;
NEW_BAL := PAST_DUE + INTEREST + PURCHASE;
PRINT (OLD_BAL, PAYMENT, PAST_DUE, INTEREST, PURCHASE,
   NEW_BAL);
OLD_BAL := 59.50;          # THIRD ACCOUNT #
PAYMENT := 29.75;
PURCHASE := 99.95;
PAST_DUE := OLD_BAL - PAYMENT;
INTEREST := PAST_DUE * 0.015;
NEW_BAL := PAST_DUE + INTEREST + PURCHASE;
PRINT (OLD_BAL, PAYMENT, PAST_DUE, INTEREST, PURCHASE,
   NEW_BAL);
```

The first statement declares all six variables used in the program. The executable statements follow. First OLD_BAL is assigned the balance (in dollars) at the beginning of the month for some account; PAYMENT is assigned the amount of payments for the month; and PURCHASE is assigned the amount of new purchases during the month. Then PAST_DUE is calculated as the amount of the old balance less payments. INTEREST is set to 1.5% of PAST_DUE and NEW_BAL is calculated as the sum of the amount past due, the interest on it, and the amount of new purchases during the month. Finally, the values of all six variables are printed out. At this point the process is repeated a second and third time, with values for a different account assigned to OLD_BAL, PAYMENT and PURCHASE. After the last statement in the list is executed, the program's execution is completed. The printed output looks like this:

```
4.40000E+01 4.40000E+01 0.00000E+00 0.00000E+00
2.50000E+01 2.50000E+01
1.96950E+02 2.50000E+01 1.71950E+02 2.57925E+00
0.00000E+00 1.74529E+02
5.95000E+01 2.97500E+01 2.97500E+01 4.46250E-01
9.99500E+01 1.30146E+02
```

If you change the results to non-exponential notation, and then round them to two decimal places, you get the answers in dollars and cents.

This is not at all a very sophisticated program. It scarcely begins to take advantage of the capabilities of a computer. In the following chapters we will show how this program can be made better and more powerful, using other statements of CS-4.

4.0

INPUT AND LOOPING: WRITING A REUSABLE PROGRAM

The biggest problem with the program in the previous section is that every account has its own set of statements. Every time you use the program you have to write new assignment statements for OLD_BAL, PAYMENT, and PURCHASE, and then recompile the whole program.

There is another deficiency: you have to write the four statements:

```
PAST_DUE := OLD_BAL - PAYMENT;
INTEREST := PAST_DUE * 0.015;
NEW_BAL := PAST_DUE + INTEREST + PURCHASE;
PRINT (OLD_BAL, PAYMENT, PAST_DUE, INTEREST, PURCHASE,
   NEW_BAL);
```

over and over again, once for each account you are processing. This means the compiler must compile these same statements repeatedly, which would be quite a waste of time if there were hundreds or thousands of accounts.

In this chapter you will see how to read and assign values from input cards, and how to execute the same statements more than once. The result will be a very short program that can process any number of accounts.

Assigning values from input cards

You don't have to assign initial values to variables by writing assignment statements in your program. Instead, you can punch the values on separate input cards, and write a READ_LINE statement that reads values from the cards and assigns them to the variables.

Input cards are not part of the CS-4 program which is compiled, and they are not read by the compiler. They are read by the translated program itself, when it is executed. They are usually placed after the cards that contain the program; in any case, the control cards that accompany the program will indicate where the input cards begin and end.

The READ_LINE statement has the same basic form as the PRINT statement:

```
READ_LINE (OLD_BAL, PAYMENT, PURCHASE);
```

When it is executed it causes an input card to be read, and causes the values punched on the card to be assigned to the respective variables in the list in parentheses. For example, if the statement above read the input card

```
59.50       29.75       99.95
```

it would have the same effect as

```
OLD_BAL := 59.50;
PAYMENT := 29.75;
PURCHASE := 99.95;
```

You are not required to punch all the input values on a single card. The READ_LINE statement will read as many cards as are necessary in order to assign one value to each of its variables. In our example, the READ_LINE statement would make the same assignments if it encountered two input cards like this:

```
59.50       29.75
99.95
```

or three input cards like this:

```
59.50
29.75
99.95
```

The only thing that matters is that the input values appear in the right order.

Input values may be punched anywhere on the input card, as long as they are separated from each other by at least one space. There is no semicolon after the last value on the input card, because the input values are not CS-4 statements.

Every time a READ_LINE statement is executed, it starts reading a new card. This means you cannot read a single input card like

```
59.50       29.75       99.95
```

by writing two statements:

```
READ_LINE (OLD_BAL, PAYMENT);
READ_LINE (PURCHASE);
```

The first READ_LINE will assign the first two values and ignore the 99.95; and then PURCHASE will be assigned a value from the next input card.

READ_LINE may be thought of as a sort of inverse of PRINT. PRINT outputs values from the program; READ_LINE inputs values to it. However, keep in mind that READ_LINE does change the values of the variables in the parentheses, while PRINT does not. Hence, only PRINT statements can contain literals. The statement

```
READ_LINE (2.0);
```

is an error, since it is meaningless to assign a new value to the literal constant 2.0.

Changing input from run to run

READ_LINE makes it unnecessary to write a separate set of assignment statements for every set of account data. Instead, you can write

```
READ_LINE (OLD_BAL, PAYMENT, PURCHASE);
```

and just provide a separate input card for each set of data. Then the accounting program will look like this:

```
VARIABLES OLD_BAL, NEW_BAL, PURCHASE, PAST_DUE,
   INTEREST, PAYMENT ARE REAL;
READ_LINE (OLD_BAL, PAYMENT, PURCHASE);     # ACCT 1 #
PAST_DUE := OLD_BAL - PAYMENT;
INTEREST := PAST_DUE * 0.015;
NEW_BAL := PAST_DUE + INTEREST + PURCHASE;
PRINT (OLD_BAL, PAYMENT, PAST_DUE, INTEREST,
   PURCHASE, NEW_BAL);
READ_LINE (OLD_BAL, PAYMENT, PURCHASE);     # ACCT 2 #
PAST_DUE := OLD_BAL - PAYMENT;
INTEREST := PAST_DUE * 0.015;
NEW_BAL := PAST_DUE + INTEREST + PURCHASE;
PRINT (OLD_BAL, PAYMENT, PAST_DUE, INTEREST,
   PURCHASE, NEW_BAL);
READ_LINE (OLD_BAL, PAYMENT, PURCHASE);     # ACCT 3 #
PAST_DUE := OLD_BAL - PAYMENT;
INTEREST := PAST_DUE * 0.015;
NEW_BAL := PAST_DUE + INTEREST + PURCHASE;
PRINT (OLD_BAL, PAYMENT, PAST_DUE, INTEREST,
   PURCHASE, NEW_BAL);
```

and to do the same work as the example in the previous chapter, it would need the following sequence of input cards:

```
44.00        44.00        44.00
196.95        25.00       0.00
59.50        29.75        99.95
```

If you want to process a different set of account data, you just run the same program again with new input cards.

The program is now basically just a declaration statement followed by a sequence of five statements -- a READ_LINE, three assignments, and a PRINT -- which is repeated over and over again. Each five-statement sequence is exactly the same; but each calculates with different values -- because each READ_LINE reads a different input card.

Changing the flow of control

Normally each executable statement in a program is executed immediately after the statement before it. In other words, the flow of control begins with the first statement, and passes to each succeeding statement until the end of the program.

However, as we have seen from the sample billing program, the same statements may actually be executed many times. One way to have statements executed repeatedly is to write them in the program repeatedly, as we did in the last section, so that control passes to them again and again. But it is much more useful to have a way of altering the flow of control, so that a statement or sequence of statements that is written once is performed over and over again. (In other cases, which you will see in the next chapter, it is desirable to alter the flow of control so that some statements are skipped over and not executed at all.)

A statement or sequence of statements that is executed repeatedly is called a loop. You can indicate that a list of statements in CS-4 is to be looped by writing

```
REPEAT
   statement;
   statement;
   . . .
   statement;
END;
```

where each "statement" is an executable statement.

The sequence of statements between REPEAT and END will be interpreted by the compiler as a loop. After each time the last statement in the sequence is executed, the flow of control passes back to the first statement in the sequence.

The word REPEAT followed by a statement list and END is called a REPEAT statement. REPEAT statements have more general and powerful uses than the simple one we have just used. However, we will postpone a general exposition of the REPEAT statement until Chapter 8.

A looping program

We can use REPEAT to rewrite the sample billing program as a simple program of about half a dozen statements:

```
VARIABLES OLD_BAL, NEW_BAL, PURCHASE, PAST_DUE, INTEREST,
   PAYMENT ARE REAL;
REPEAT
   READ_LINE (OLD_BAL, PAYMENT, PURCHASE);
   PAST_DUE := OLD_BAL - PAYMENT;
   INTEREST := PAST_DUE * 0.015;
   NEW_BAL := PAST_DUE + INTEREST + PURCHASE;
   PRINT (OLD_BAL, PAYMENT, PAST_DUE, INTEREST,
      PURCHASE, NEW_BAL);
END;
```

Let us assume that each input card contains three input values. When execution of this program begins, the first executable statement in the loop -- the READ_LINE statement -- is performed first. It assigns values to OLD_BAL, PAYMENT, AND PURCHASE from the first input card. Subsequent statements calculate PAST_DUE, INTEREST, and NEW_BAL, and print out the values of all six variables.

The PRINT statement is the last statement of the loop. After it is executed, control is returned to the loop's first statement -- the READ_LINE statement. The READ_LINE statement is thus executed again. It assigns new values to its three variables from the second input card. The calculations and printout are then repeated using these new values. Finally, control is again passed back to the READ_LINE statement, which reads in new values from still another card.

In short, the five statements from READ_LINE to PRINT are repeated indefinitely, as long as there are new input cards to be read. There will be one set of values printed for every card that is read. This same program can be run many times with different sets of input cards, and the number of input cards can vary from run to run.

Loops are essential to writing efficient programs that process input data. In almost any program of this sort you will find at least one loop.

5.0

TESTING CONDITIONS: THE IF STATEMENT

Our sample billing program is now short and efficient. If anything, it is too short -- it fails to take into account two important possibilities:

1) What if an account is overpaid? The program as it is now written computes a negative interest in this case. However, in most revolving charge accounts of this type, the interest on an overpayment is zero.

2) What happens when there are no more accounts to be processed? As the program is now written, it continues looping indefinitely. On some installations, the program is automatically terminated when you run out of input cards, but it is not good practice to rely on this.

These two situations demonstrate an important fact about computer programming: you must explicitly provide for all possibilities. If you want to compute interest differently in some cases, or if you want to tell which input card is last, you have to write statements to do it.

How is this done? In both cases, you want to test some condition -- whether there is an overpayment, whether the last card has been reached -- and perform different actions depending on the result of the test. The CS-4 IF statement, which is described in this chapter, enables you to make such tests.

Writing equalities and inequalities

Since so far you have been working only with numerical values, the only sorts of conditions you can test are numerical equality and inequality. In order to test an equality or inequality in CS-4, you must first write it in CS-4 notation.

CS-4 has six infix comparison operators, whose forms resemble familiar symbols of arithmetic:

=	equal
~=	not equal
>	greater than
<	less than
>=	greater than or equal
<=	less than or equal

These symbols are used to form comparison expressions, such as:

Expression	Expression is true when
A = B + 1	the value of A equals the value of B plus 1
C + D ~= 5.1	the value of the expression C + D is not equal to the value 5.1
COUNT > LIMIT	the value of COUNT is greater than that of LIMIT
B*B - 4*A*C >= 0	the value of the expression B*B - 4*A*C is greater than or equal to 0

As you see, at any given time during execution of a program, a comparison expression may be either true or false, depending on the values of the variables in it. There may be arithmetic operators in a comparison expression, in which case the arithmetic operations are performed before the comparison operation.

Comparison operators in CS-4 are used only to write conditions to be tested; they never change the values of their operands. To cause a new value to be assigned to a variable, you use the assignment operator. Hence it is correct to write either of the two expressions:

```
2 = A
A = 2
```

both of which are true only if A has the value 2. On the other hand, the expression

```
A := 2
```

does not have a truth value; it assigns the value 2 to the variable A. And its reverse:

```
2 := A
```

is an error -- one cannot assign a new value to the literal 2.

Testing equalities and inequalities

Now you are ready to write a statement that tests a comparison expression, and that says what to do if the expression is true. This is the IF statement, which is written generally like this:

```
IF comparison THEN statement;
```

where "comparison" is some comparison expression, and "statement" is any CS-4 executable statement. When the IF statement is executed, it first evaluates the comparison expression. The statement following THEN is executed next if the comparison expression is true; if the comparison is false, it is not executed at all.

As an example, consider

```
IF COUNT = 0 THEN PRINT (ALTITUDE);
```

This statement causes PRINT (ALTITUDE) to be executed when the value of COUNT is 0; otherwise it has no effect. In either case, the program then continues with the next statement. We say that the PRINT statement is conditionally executed. That is, the flow of control is passed to it only when the condition -- COUNT having the value 0 -- is true.

IF only tests a condition -- it does not change the values of any of the variables in the comparison expression. In our example, COUNT will have the same value after the IF that it did before.

IF is an example of compound statement: a statement that contains another statement. CS-4 has several other compound statements -- including the general forms of REPEAT and BEGIN -- which will be introduced in succeeding chapters.

Using IF statements to improve the billing program

Now you can test for an overpayment, by using an IF statement of the form

```
IF PAYMENT > OLD_BAL THEN ...
```

How do you test for the last card? One way is by using an additional input -- an account number, for example. You use only positive numbers for real accounts, and zero as a dummy account number to indicate the last card. Then a test of the form

```
IF ACCT_NO <= 0 THEN ...
```

where ACCT_NO holds the account number value, will identify the last card.

The program can now be written like this:

```
VARIABLES ACCT_NO, OLD_BAL, NEW_BAL, PURCHASE, PAST_DUE,
   INTEREST, PAYMENT ARE REAL;
REPEAT

   READ_LINE (ACCT_NO, OLD_BAL, PAYMENT, PURCHASE);
   IF ACCT_NO <= 0 THEN TERMINATE;
   PAST_DUE := OLD_BAL - PAYMENT;
   IF PAYMENT > OLD_BAL THEN PAST_DUE := 0;
   INTEREST := PAST_DUE * 0.015;
   NEW_BAL := OLD_BAL - PAYMENT + INTEREST + PURCHASE;
   PRINT (ACCT_NO, OLD_BAL, PAYMENT, PAST_DUE, INTEREST,
      PURCHASE, NEW_BAL);
END;
```

The first IF statement checks ACCT_NO. If it is not positive, control passes to a type of statement we have not used before:

```
TERMINATE;
```

which causes the program to stop executing immediately.

If ACCT_NO is positive, control passes to the next statement, which is the assignment of a value to PAST_DUE. Then comes another IF, which checks whether PAYMENT is greater than OLD_BAL. If it is, the value of PAST_DUE is changed to zero; otherwise, PAST_DUE is left the same.

Note that the order of the two statements

```
PAST_DUE := OLD_BAL - PAYMENT;
IF PAYMENT > OLD_BAL THEN PAST_DUE := 0;
```

is significant. It would have been wrong to write

```
IF PAYMENT > OLD_BAL THEN PAST_DUE := 0;
PAST_DUE := OLD_BAL - PAYMENT;
```

With the statements in this order, PAST_DUE is still set to zero when there is an overpayment. But then control passes to the second statement, which incorrectly resets PAST_DUE to OLD_BAL - PAYMENT.

Note also the importance of choosing the comparison expression carefully. The program as it now stands will terminate when an account number is negative. Had we written

```
IF ACCT_NO = 0 THEN TERMINATE;
```

negative account numbers would be accepted. In practice, we would expect only non-negative account numbers: positive ones for real accounts, and zero to signal the end of the inputs. But it is possible that some error would produce a negative account number; in that case we would want to stop execution so that the error could be corrected before the rest of the accounts were processed. We have therefore chosen to write the test with <=. (In the same vein, we might want to add a test that PAYMENT is non-negative, if a negative payment is definitely an error.)

Adding an ELSE clause

Our revised billing program in the previous section was somewhat inefficient in checking for an overpayment. We used two statements:

```
PAST_DUE := OLD_BAL - PAYMENT;
IF PAYMENT > OLD_BAL THEN PAST_DUE := 0;
```

When there is an overpayment, the first statement sets PAST_DUE to a negative value; then the second statement resets it to zero. Thus, for overpaid accounts, the first assignment serves no purpose, because it is immediately overridden by the assignment in the IF statement.

An alternative way to write the program, so that only one value is assigned to PAST_DUE in any case, uses two IFs:

```
IF PAYMENT > OLD_BAL THEN PAST_DUE := 0;
IF PAYMENT <= OLD_BAL THEN PAST_DUE := OLD_BAL - PAYMENT;
```

In this construction, however, the conditional expression in the second IF is redundant. It is true only when the conditional in the first IF is false, and is false only when the conditional in the first IF is true. We could be still more efficient if there were some way to append two statements to the first IF: one to be performed when the condition is true, the other when it is false. This can be done by adding an ELSE clause:

```
IF PAYMENT > OLD_BAL THEN PAST DUE := 0;
ELSE PAST_DUE := OLD_BAL - PAYMENT;
```

The effect of this statement is the same as that of the two IFs above.

In general, an ELSE clause -- consisting of ELSE followed by a CS-4 executable statement -- may be part of any IF statement. When an IF has both THEN and ELSE clauses, the THEN clause always comes first. There need be no semicolon between the two clauses (though of course there must be a semicolon at the end of the entire IF statement); but in this Primer we always do put a semicolon after the THEN clause, because the statement reads a bit easier that way.

Each time control passes to an IF statement, either the THEN clause or the ELSE clause (if any) is executed -- the former if the condition is true, the latter if it is false. Control then passes to the statement following the IF, unless the clause executed was a TERMINATE.

IF statements within IF statements

The statement in a THEN or ELSE clause may be another IF statement. This is often quite useful when there are more than two possibilities to choose from.

For example, in some states the law limits interest rates in steps. The legal amount might be 1.5% on balances not more than $500, 1.25% on any part of a balance over $500 but not more than $1000, and 1% on any part of a balance over $1000. To compute the interest you could write

```
IF PAST_DUE <= 500 THEN INTEREST := PAST_DUE * 0.015;
ELSE
   IF PAST_DUE <= 1000 THEN INTEREST := PAST_DUE * 0.0125
      + 1.25;
   ELSE INTEREST := PAST_DUE * 0.01 + 3.75;
```

When an IF statement appears in a THEN clause, the result may appear ambiguous:

```
IF X >= 0 THEN
   IF Y >= 0 THEN Z := Y * X;
   ELSE Z := 0;
```

The compiler must decide which IF statement the ELSE clause is part of. The rule it follows is this: an ELSE clause is always part of the closest IF which does not already have an ELSE clause. In our example, therefore, the ELSE is interpreted as part of the second IF.

6.0

TWO NEW MODES: BOOLEAN AND INTEGER

Thus far we have dealt only with variables of REAL mode, whose values are numerical values. In this chapter we introduce another numerical-valued mode, INTEGER. We also introduce the mode BOOLEAN, which allows only two values which represent "true" and "false".

BOOLEAN mode

All the arithmetic expressions we have seen so far have values of REAL mode. But how about the comparison expressions described in the previous chapter? Do they have a value the same way arithmetic expressions do? If so, what mode is it?

Comparison expressions do have values. They are of a mode called BOOLEAN. Whereas REAL represents all possible numerical values, BOOLEAN represents all possible truth values. So there are only two possible values of mode BOOLEAN -- one that represents "true", and one that represents "false". These are all the values that are needed, since every comparison expressions is either true or false.

You can use the mode name BOOLEAN in ways analogous to the uses of the mode name REAL. You can define BOOLEAN-valued variables in declaration statements:

```
VARIABLE COMPARE IS BOOLEAN;
VARIABLES SWITCH1, SWITCH2 ARE BOOLEAN;
```

Variables declared in this way can only represent one of the two truth values.

There are two BOOLEAN literals, written

```
FALSE
TRUE
```

which can be assigned to BOOLEAN variables:

```
COMPARE := TRUE;
SWITCH1 := FALSE;
```

You can also make the assignments with a **READ_LINE** instruction

```
READ_LINE (COMPARE, SWITCH1);
```

that reads an input card punched:

```
TRUE    FALSE
```

BOOLEAN variables can be assigned the values of comparison expressions, in the same way that REAL variables can be assigned the results of arithmetic expressions. For example:

```
SWITCH2 := INTEREST = 0;
COMPARE := B * B >= 4 * A * C;
```

SWITCH2 is set to TRUE when INTEREST is zero, and FALSE when it is non-zero. COMPARE is set to FALSE when B*B is less than 4*A*C, and to TRUE otherwise. (Note again that variables in the comparison expressions -- INTEREST, B, A, C -- are not changed in value by the comparison operators. Only the variables to the left of an assignment operator -- SWITCH2 and COMPARE -- are assigned new values.)

BOOLEAN values may be used as operands to the comparison operators = and ~=. For example

```
SWITCH1 ~= SWITCH2
```

has the value TRUE when SWITCH1 and SWITCH2 have different values (one TRUE, the other FALSE), and has the value FALSE when they have the same value (both TRUE, or both FALSE). (Using the terminology of formal logic, ~= is exclusive OR, and = is equivalence.)

BOOLEAN values cannot be used as operands to the other comparison operators or to arithmetic operators. Expressions such as

```
SWITCH1 + SWITCH2
COMPARE <= INTEREST
```

will be rejected as errors by the compiler. The compiler will also

reject any attempt to assign a BOOLEAN value to a REAL variable, or a REAL value to a BOOLEAN variable:

```
INTEREST := FALSE;
COMPARE := 1;
```

Logical operators

There are six logical operators in CS-4 wich accept BOOLEAN values as operands. They perform the common truth functions:

`&` logical AND (infix) -- A & B has the value TRUE if both A and B have the value TRUE; otherwise A & B has the value FALSE

`|` logical inclusive OR (infix) -- A | B has the value TRUE if A or B or both have the value TRUE; otherwise A | B has the value FALSE

`XOR` logical exclusive OR (infix) -- A XOR B has the value FALSE if both A and B have the value TRUE or if both have the value FALSE; otherwise A XOR B has the value TRUE

`NAND` logical NAND (infix) -- A NAND B has the value FALSE if both A and B are TRUE; otherwise A NAND B has the value TRUE

`NOR` logical NOR (infix) -- A NOR B has the value TRUE if both A and B are FALSE; otherwise A NOR B is FALSE

`~` logical NOT (prefix) --~A has the value TRUE if A has the value FALSE, and ~A has the value FALSE if A has the value TRUE

We will call expressions formed from logical operators boolean expressions, because they always have values of BOOLEAN mode.

BOOLEAN variables, comparison expressions, or some combination of them may be used in boolean expressions. For example:

```
X = 0 & Y = 0
INTEREST >= 1.25 & (Switch1 | ~SWITCH2)
~(SWITCH1 & SWITCH2 & COMPARE)
```

The first expression has the value TRUE only if both X and Y have the value 0. The second has TRUE only if INTEREST is at least 1.25, and if either SWITCH1 is TRUE or SWITCH2 is FALSE. The third has FALSE only if SWITCH1, SWITCH2 and COMPARE all are TRUE.

When logical operators are used together in expressions, parentheses may be used to force evaluation in a certain order. When parentheses do not dictate otherwise, the order of evaluation is determined by operator precedence: ~ is applied before & and NAND, which are applied before |, NOR, and XOR. That is

```
~A & ~B | C
```

is interpreted as if it were parenthesized

```
((~A) & (~B)) | C
```

Unless parentheses alter the order, logical infix operators are applied after all arithmetic and comparison operators. (A complete table of precedences of all operators used in this volume is given in Appendix B, p. 290.)

Only BOOLEAN values may be operands to logical operators. Use of REAL-valued operands, such as

```
PAST_DUE & INTEREST > 0
ACCT_NO | 1
```

will be rejected as errors by the compiler.

Boolean expressions in IF statements

We can now generalize one of the rules for writing IF statements. The expression which is placed between IF and THEN may be any BOOLEAN-valued expression. For example, these are valid IF statements:

```
IF SWITCH1 THEN SWITCH2 := TRUE;

IF (X < 0 & Y < 0) THEN Z := X * Y;
ELSE Z := 0;

IF ACCT_NO = 0 & COMPARE THEN TERMINATE;
ELSE PRINT (ACCT_NO);
```

If the expression evaluates to TRUE, the statement following THEN is executed. If the expression evaluates to FALSE, the statement following ELSE, if any, is executed.

Using BOOLEAN variables and expressions

To demonstrate some uses of BOOLEAN mode and logical operators, we add two new requirements to the sample billing program:

1) Some accounts will be interest-free. This will be indicated by a BOOLEAN value on each input card, which will be TRUE when the account is interest-free, and FALSE otherwise.

2) The number of accounts that had interest charged on them will be printed out after all the input has been read.

```
VARIABLES ACCT_NO, OLD_BAL, NEW_BAL, PURCHASE, PAST_DUE,
   INTEREST, PAYMENT, PAST_COUNT ARE REAL;
VARIABLE INTEREST_FREE IS BOOLEAN:
PAST_COUNT := 0;
REPEAT
   READ_LINE (ACCT_NO, OLD_BAL, PAYMENT, PURCHASE,
      INTEREST_FREE);
   IF ACCT_NO <= 0 THEN PRINT (PAST_COUNT);
   IF ACCT_NO <= 0 THEN TERMINATE;
   INTEREST_FREE := INTEREST_FREE | PAYMENT >= OLD BAL;
   IF INTEREST_FREE THEN PAST_DUE := 0;
   ELSE PAST_DUE := OLD_BAL - PAYMENT;
   INTEREST := PAST_DUE * 0.015;
   NEW_BAL := OLD_BAL - PAYMENT + INTEREST + PURCHASE;
   PRINT (ACCT_NO, OLD_BAL, PAYMENT, PAST_DUE, INTEREST,
      PURCHASE, NEW_BAL);
   IF ~INTEREST_FREE THEN PAST_COUNT := PAST_COUNT + 1;
END;
```

The BOOLEAN input value is assigned to INTEREST_FREE. After the test of ACCT_NO, we assign a new value to INTEREST_FREE with this statement:

```
INTEREST_FREE := INTEREST_FREE | PAYMENT >= OLD_BAL;
```

The new value of INTEREST_FREE is FALSE only when its original input value was FALSE, and PAYMENT is less than OLD_BAL. In other words, this statement modifies INTEREST_FREE so that it is FALSE only when there is some interest to be paid, and is TRUE otherwise. (It is entirely proper to use a variable name on both sides of an assignment operator. The old value of the variable is always used in computing the value of the right-hand expression before the new value is assigned.)

PAST_DUE is computed if INTEREST_FREE is FALSE, and is set to zero if it is TRUE. The loop then continues as before. A new IF has been added at the end of the loop which increases the value of PAST_COUNT by 1 only if INTEREST_FREE is FALSE. (Again, we have a variable on both sides of the assignment operator. It assigns PAST_COUNT a new value one greater than its old value.) Notice that PAST_COUNT was set to zero before the loop began.

As before, the loop continues indefinitely until a non-positive value of ACCT_NO signals the last card:

```
IF ACCT_NO <= 0 THEN PRINT (PAST_COUNT);
IF ACCT_NO <= 0 THEN TERMINATE;
```

Then PAST_COUNT is printed out, and the program is terminated by TERMINATE.

INTEGER mode

There is another mode in CS-4 whose values are numerical. It is called INTEGER. As the name suggests, variables of this mode can only represent integral values (0, 1, 2, 3, ... and -1, -2, -3, ...).

Of course, REAL variables can also be used to represent integral numerical values. Why does CS-4 include a separate INTEGER mode? There are a number of reasons:

1) Because of the problems of representing a large range of numerical values in a finite space in a computer, the representation of REAL values may be slightly inexact. (This fact is related to the limit on the number of significant digits.) INTEGER values, which are more limited in range, are always represented exactly. Very often a program has variables that are only used for counting (such as

PAST_COUNT in the billing program) or numbering (such as ACCT_NO). These variables always have integral values. They are often used in comparisons, where exactness is important. So variables of this sort are usually declared of INTEGER rather than REAL mode.

2) Because INTEGER values do not have decimal points to be kept track of, some machines can store or manipulate them more efficiently than REAL values. Thus some programs can be run more economically using INTEGERs rather than REALs.

3) INTEGER mode can be used to force REAL values to be rounded off to the nearest integer. We will show how this can be done, and how it is useful, in a later section.

Variables of INTEGER mode are declared in the same way as other variables:

```
VARIABLE PAST_COUNT IS INTEGER;
VARIABLES I1, I2, I3 ARE INTEGER;
```

Literals without decimal points and without exponent parts are INTEGER-literals; they can stand for values with INTEGER representations. Literals with decimal points or exponent parts are REAL-literals, whose values must be stored with the REAL representation.

Expressions and assignments using only INTEGER values

The arithmetic operators +, -, and * produce an INTEGER value from two INTEGER operands, just as they produce a REAL value from two REAL operands. If an expression uses only +, -, and *, if all the variables in it are INTEGER, and if all literals are INTEGER-literals, then the value of the expression is of INTEGER mode. For instance, assuming

```
VARIABLES I, J, K ARE INTEGER;
```

then the following:

```
I + J
I - 1
I * -K
100 - J * (1 - K)
```

are all proper INTEGER-valued expressions.

The assignment operator is used with INTEGER values in the same way as with REAL values:

```
I := J * 2 + K
J := J + 1
```

The comparison operators also can use INTEGER operands in the same way as REAL ones:

```
I ~= 1
J > I * K
```

The latter expression, for instance, has the BOOLEAN value TRUE if the value of J is greater than the value of I * K, and has FALSE otherwise.

If you want to take advantage of the computer's ability to handle INTEGER values more exactly and efficiently than real ones, you should as much as possible:

1) use INTEGER-valued expressions;

2) assign INTEGER values to INTEGER variables;

3) use only INTEGER values in comparison expressions.

Mixing INTEGER and REAL in arithmetic expressions

It is permissable, and often convenient, to use +, -, or * with one INTEGER operand and one REAL operand. In such a case, the representation of the INTEGER operand is first converted to REAL representation; then the operator can be applied to two REALs, producing a REAL. It is almost always possible to perform a conversion from INTEGER to REAL representation, without changing the operand's numerical value -- because the values represented by INTEGERs are a subset of the values represented by REALs.

The only exception comes when an INTEGER has more significant digits than a REAL can represent; an attempt to convert an over-precise INTEGER of this sort to REAL would be in error. (The term conversion is used generally in CS-4, to refer to any procedure for transforming a value of one mode to a value of another mode.)

The other two arithmetic operators, / and **, always produce a REAL value, even when applied to two INTEGER operands. Any INTEGER operands to / or ** are converted to REAL before the operation is applied.

Taking the behavior of the five arithmetic operands together, we can say that any arithmetic expression is of REAL mode if it:

1) contains at least one REAL variable; or

2) contains at least one REAL-literal; or

3) uses the / or ** operator -- even with two INTEGER operands.

For example, assume

```
VARIABLES I1, I2 ARE INTEGER;
VARIABLES R1, R2 ARE REAL;
I1 := 1;
I2 := 3;
R1 := 2.7;
R2 := -4.9;
```

then we can write

Expression	Value (REAL mode)
-I1 + R2	-5.9
I2 * 3.0	9.0
R1 / 3	0.9
I1 + I2 + (R1 * 3)	12.1
I1 / I2	0.33333333
I2 ** I1	3.0

Any of these expressions could be assigned to a REAL-mode variable.

Conversions from INTEGER to REAL sometimes take time, so expressions with all REAL variables are often more efficient than mixed ones. There is no penalty, however, for using INTEGER-literals in a REAL-valued expression. When the compiler encounters an INTEGER-literal in a place where an INTEGER value would have to be converted to a REAL, it uses a REAL representation for the literals. Thus writing

```
3. * (R1 ** 2.)
```

is no more efficient than the same expressions with decimal points dropped:

```
3 * (R1 ** 2)
```

Mixed assignments -- INTEGER to REAL

An INTEGER-mode value may be assigned to a REAL-mode variable. The INTEGER value is automatically converted to the corresponding REAL representation before the assignment is made. For instance, given the declarations and assignments of the previous section, the following all assign the same value to R2:

```
R2 := I1 + I2;
R2 := I1 + 3;
R2 := 4;
R2 := 4.0;
```

In the first two examples, the expression on the right hand side yields an INTEGER value, which must be converted to a REAL before it can be stored in R2. In the latter two examples, the literals are stored with REAL representations, and no conversion is necessary.

None of these assignments changes the mode of R2. An assignment statement never changes the mode of any variable in it.

Mixed assignments -- REAL to INTEGER

It is permissible to write an assignment of a REAL-mode value to an INTEGER-mode variable. When such an assignment is executed, the REAL value is converted to INTEGER by rounding to the closest INTEGER value before the

assignment is made. That is, if the absolute value of the REAL has a fractional part >= 0.5, it is converted to the INTEGER with the next larger absolute value. Otherwise, the fractional part is dropped.

To make some examples, assume

```
VARIABLE I IS INTEGER;
VARIABLES A1, A2, A3 ARE REAL;
A1 := 2.3;
A2 := 4.5;
A3 := 3.7;
```

Then

	Value (REAL) of right operand	Value (INTEGER) assigned to I
I := A1;	2.3	2
I := A2;	4.5	5
I := A3;	3.7	4
I := -A2;	-4.5	-5
I := -A3;	-3.7	-4
I := A2 * 2.0;	9.0	9
I := A1 + A2 + A3;	18.95	19
I := 5 / 3;	1.66666666	2

Again, the mode of the left-hand variable, I, is not changed by any of the assignments. Rather, the value of the right-hand side is converted to an INTEGER value before the assignment is made.

Occasionally the rounding triggered by a REAL to INTEGER assignment is especially useful. For example, our billing program sometimes produces values of INTEREST and NEW_BAL in fractions of a cent; we would prefer to round these values off to the nearest cent.

We can easily change the program to perform rounding. The variables must be declared INTEGER instead of REAL:

```
VARIABLES ACCT_NO, OLD_BAL, NEW_BAL, PURCHASE, PAST_DUE
   INTEREST, PAYMENT, PAST_COUNT ARE INTEGER;
```

and the input must be in cents instead of dollars:

```
Old input form:  705  59.50  29.75  99.95  FALSE
New input form:  705  5950   2975   9995   FALSE
```

When INTEREST is calculated

```
INTEREST := PAST_DUE * 0.015;
```

the right hand side has the REAL value 44.625, because of the presence of the REAL literal 0.015 in the expression. However, since INTEREST is of INTEGER mode it is assigned the rounded value 45. The calculation of NEW_BAL then makes use of this rounded value:

```
NEW_BAL := OLD_BAL - PAYMENT + INTEREST + PURCHASE;
```

The resulting output is uniformly in whole cents:

Old output form:

```
 7.05000E+02 5.95000E+01 2.97500E+01 2.97500E+01
 4.46250E-01 9.99500E+01 1.30146E+02
```

New output form:

```
 705 5950 2975 2975 45 9995 13015
```

Comparing INTEGER and REAL values

It is legal to write a comparison expression in which one operand has a REAL value and the other has an INTEGER value. The INTEGER operand value is converted to the corresponding REAL value before the comparison is made.

On some computers, mixed comparisons may yield unintended results, because of the inexactness in the way REAL values are stored. For example, you would expect the sequence of statements

```
VARIABLE I IS INTEGER;
VARIABLES R1, R2, ARE REAL;
I := 9;
R1 := 1.8;
R2 := 5.0 * R1;
IF I <= R2 THEN ...
```

to compute the value TRUE for I <= R2. However, on some machines R2 would be left with a slightly inexact value like

```
8.99999999
```

This is close enough to 9.0 for almost any calculational purpose. Yet it causes I <= R2 to have the value FALSE, because the value of I will be converted to the REAL representation of 9, which is greater than 8.99999999.

Actually, you can get into the same sort of trouble comparing two REALs, when they have nearly the same values. For example, if I above were declared of REAL mode, I <= R2 would still be computed as FALSE.

Because of the imprecise representation of REALs, you should be especially cautious when writing comparisons with one or two REAL operands when the values are expected to be very close.

Declaring variables of several modes in a single declaration statement

If you are declaring variables of different modes:

```
VARIABLE I IS INTEGER;
VARIABLES A1, A2, A3 ARE REAL;
```

you do not have to write two declaration statements. You can put both declarations in one statement, separated by commas:

```
VARIABLE I IS INTEGER,
   A1, A2, A3 ARE REAL;
```

The meaning is the same either way.

Initializing in a declaration

Often you will want to assign a value to a variable at the beginning of a program. For example, we give PAST_COUNT in the billing program an initial value of 0:

```
VARIABLE PAST_COUNT IS INTEGER;
PAST_COUNT := 0;
```

These statements can be combined by using an initialization operator, which is written as two colons followed by an equal sign (::=). The declaration is placed to the left of the initialization operator, and the intial value to its right:

```
VARIABLE PAST_COUNT IS INTEGER ::= 0;
```

This statement both declares the mode of PAST_COUNT and causes it to be assigned 0 before each time the program is executed. If an initial value is specified for a declaration of several variables:

```
VARIABLES SWITCH1, SWITCH2, SWITCH3 ARE BOOLEAN ::= TRUE;
```

The initial value is assigned to each of them before execution. In this case, SWITCH1, SWITCH2 and SWITCH3 are all assigned TRUE.

CONSTANT declarations

Sometimes a name is used in a program to represent a fixed value. Such a name may be declared as a variable, of course. But it may also be declared as a constant. A declaration statement for a constant is almost the same as one for a variable; but the word VARIABLE or VARIABLES is replaced by CONSTANT:

```
CONSTANT INTEREST_RATE IS REAL ::= 0.015;
```

A constant may not be assigned a value in an executable statement, such as an assignment statement or a READ_LINE. It may only be given a value with the initialization operator.

CONSTANT declarations permit you to give a name to bulky literals that are used in many places. For instance, you might write

```
CONSTANT PI IS REAL ::= 0.314159E+01;
```

to avoid repeating the long literal many times in a program. The name PI is also a mnemonic that makes the significance of the value clearer to the reader. And because PI is a constant and not a variable, the compiler will report an error if an attempt is made to assign it some value elsewhere in the program.

7.0

STATEMENT BLOCKS

The IF statement as we have described it may have only one simple statement (like assignment or TERMINATE) in a THEN or ELSE clause. This often proves to be a cumbersome limitation. Even our small billing program could be more efficient in a number of ways if several simple statements could be executed by a single THEN or ELSE clause.

The most obvious example is the pair of IF statements which test ACCT_NO:

```
IF ACCT_NO <= 0 THEN PRINT (PAST_COUNT);
IF ACCT_NO <= 0 THEN TERMINATE;
```

The same condition has to be tested twice, because each IF can only have one statement in its THEN clause. For a similar reason, we have to test INTEREST_FREE twice:

```
IF INTEREST_FREE THEN PAST_DUE := 0;
ELSE PAST_DUE := OLD_BAL - PAYMENT;
   . . .
IF ~INTEREST_FREE THEN PAST_COUNT := PAST_COUNT + 1;
```

This redundancy could be avoided if

```
PAST_COUNT := PAST_COUNT + 1;
```

could be added to the ELSE clause of the first IF.

We also waste some time by computing

```
INTEREST := PAST_DUE * 0.015;
```

even when PAST_DUE has just been set to zero by the preceding IF statement. It would be more efficient to execute

```
PAST_DUE := 0;
INTEREST := 0;
```

with a single THEN clause, or

```
PAST_DUE := OLD_BAL - PAYMENT;
INTEREST := PAST_DUE * 0.015;
```

with a single ELSE clause.

These problems are handled in CS-4 by grouping statements together into what are known as BEGIN blocks. In this chapter we give a description of BEGIN blocks, and demonstrate their use with IF statements.

BEGIN blocks

A BEGIN block consists of the word BEGIN, followed by any number of CS-4 statements of any type, followed by END. For instance, this is a BEGIN block:

```
BEGIN;
   PAST_DUE := 0;
   INTEREST := 0;
END;
```

It is usually clearest and most convenient to punch each statement of a BEGIN block on a separate card, but it is not necessary to do so:

```
BEGIN; PAST_DUE := 0; INTEREST := 0; END;
```

The semicolon after BEGIN is optional, as is the semicolon between the last statement of the block and END. However, it is a good idea to leave them both in, for uniformity.

A BEGIN block is treated as a single executable statement. It may appear anywhere a single statement may appear -- in particular, it may be used in a THEN or ELSE clause.

How is a BEGIN block executed? By executing the statements within it, in sequence, beginning with the first one. Execution of the block concludes, normally, when control passes beyond the last statement in the list to the boundary statement END. You can thus think of the statements within a BEGIN block as a sort of "sub-program" within the main program. Executing the block means executing this "sub-program", from beginning to end. (However, if TERMINATE is executed within the block, execution does not continue to END -- because TERMINATE terminates the entire program immediately, even from within a BEGIN block.)

Non-executable statements, such as declarations, may also appear in a BEGIN block. However, declarations are interpreted slightly differently when they appear within a block, so you should keep all of your declarations outside of BEGIN blocks for the time being.

Using BEGIN blocks in IF statements

Our billing program can now be rewritten more efficiently with BEGIN blocks in each THEN and ELSE clause:

```
VARIABLES ACCT_NO, OLD_BAL, NEW_BAL, PURCHASE, PAST_DUE,
   INTEREST, PAYMENT ARE INTEGER,
   PAST_COUNT IS INTEGER ::= 0,
   INTEREST_FREE IS BOOLEAN;
REPEAT
   READ_LINE (ACCT_NO, OLD_BAL, PAYMENT, PURCHASE,
      INTEREST_FREE);
   IF ACCT_NO <= 0 THEN
      BEGIN;
         PRINT (PAST_COUNT);
         TERMINATE;
      END;
   IF INTEREST_FREE | PAYMENT >= OLD_BAL THEN
      BEGIN;
         PAST_DUE := 0;
         INTEREST := 0;
      END;
   ELSE
      BEGIN;
         PAST_DUE := OLD_BAL - PAYMENT;
         INTEREST := PAST_DUE * 0.015;
         PAST_COUNT := PAST_COUNT + 1;
      END;
   NEW_BAL := OLD_BAL - PAYMENT + INTEREST + PURCHASE;
   PRINT (ACCT_NO, OLD_BAL, PAYMENT, PAST_DUE, INTEREST,
      PURCHASE, NEW_BAL);
END;
```

Only one IF is needed to test ACCT_NO. When ACCT_NO is non-positive, the BEGIN block in the THEN clause prints PAST_COUNT and then terminates the program. Otherwise, the THEN clause is skipped over, and its BEGIN block is not executed.

The second IF executes either the BEGIN block following THEN or the one following ELSE. Both of these blocks always terminate by passing control to an END. So after either block is executed, control is passed in the normal manner, to the statement which follows the IF -- the assignment to NEW_BAL.

Note that we are now able to move the boolean expression

```
INTEREST_FREE | PAYMENT >= OLD_BAL
```

inside the second IF, because we only have to test its value once. Previously, we had to assign the value of this expression to INTEREST_FREE, so that we could test its value twice without having to compute the expression twice.

Finally, notice that END appears three times in the program -- once to conclude each of the three BEGIN blocks, and once after the last statement in the REPEAT loop. Since the first three ENDs are paired with BEGINs, only the final END marks the last statement to be repeated. One must keep careful track of which END concludes which group of statements. That is another reason for maintaining a consistent system of indentation.

Nested BEGINs and IFs

It is often useful to write an IF statement within a BEGIN block, or to write a BEGIN block for one or both clauses of an inner IF. In general, you can nest IFs within BEGINs, and BEGINs within IFs, to any degree.

As a practical example, suppose that when INTEREST_FREE is true we do want to calculate PAST_DUE, but INTEREST is still to be set to zero, and PAST_COUNT is not to be incremented. We then need to write:

```
IF PAYMENT >= OLD_BAL THEN
   BEGIN;
      PAST_DUE := 0;
      INTEREST := 0;
   END;
ELSE
   BEGIN;
      PAST_DUE := OLD_BAL - PAYMENT;
      IF INTEREST_FREE THEN INTEREST := 0;
      ELSE
         BEGIN;
            INTEREST := PAST_DUE * 0.015;
            PAST_COUNT := PAST_COUNT + 1;
         END;
   END;
```

In this case, an IF statement is nested in a BEGIN block in an ELSE clause. But we can do exactly the same work by writing the IF this way:

```
IF PAYMENT < OLD_BAL THEN
   BEGIN;
      PAST_DUE := OLD_BAL - PAYMENT;
      IF INTEREST_FREE THEN INTEREST := 0;
      ELSE
         BEGIN;
            INTEREST := PAST_DUE * 0.015;
            PAST_COUNT := PAST_COUNT + 1;
         END;
   END;
ELSE
   BEGIN;
      PAST_DUE := 0;
      INTEREST := 0;
   END;
```

In this case, the nested IF is in a THEN clause. Notice how the indentation in the example makes the structure of the program apparent.

8.0

LOOP STATEMENTS

You have already seen how to loop a group of statements indefinitely by using a REPEAT statement. In each of these loops there was a conditionally executed TERMINATE, such as

```
IF ACCT_NO < 0 THEN TERMINATE;
```

which eventually stopped the looping by terminating the whole program.

In this chapter we re-introduce REPEAT in a more general way. We will show how you can terminate a REPEAT loop without terminating the whole program, by adding qualifying phrases to the REPEAT statement. In the process, we will show a few new applications for loops within programs.

The form of the REPEAT statement

A REPEAT statement (or loop statement) is a compound statement, written like this:

```
qualifying-phrase-list REPEAT statement-list END
```

where "statement-list" is a sequence of one or more executable statements, with semicolons between them if there are more than one. The "statement-list" part is also called the loop.

The "qualifying-phrase-list" is optional. When it is omitted, the result is a simple REPEAT statement like the ones we have already seen:

```
REPEAT statement-list END
```

In a simple REPEAT statement, the loop is executed repeatedly, without limit. In earlier examples, we terminated the loop (along with the entire program) with a statement which contained a TERMINATE; we have introduced no other way to make a simple REPEAT stop looping.

It is important that every REPEAT statement you write have some way of terminating. A REPEAT that has no way of terminating creates what is called an infinite loop. In theory, an infinite loop goes on being executed forever, or until an error condition occurs -- such as

execution of READ_LINE when there are no input cards left. In practice, all programs have a time limit after which they are automatically terminated. Even so, an infinite loop can cause a considerable waste of computer time.

Qualifying a REPEAT statement with an UNTIL phrase

An alternative way to assure the termination of a REPEAT statement is to qualify it with an UNTIL phrase. A REPEAT with an UNTIL phrase has this form:

```
UNTIL condition REPEAT statement-list END
```

where "condition" is a BOOLEAN-valued expression. The "condition" is evaluated after every repetition of the loop. If it is FALSE, the loop is executed again. If it is TRUE, the REPEAT statement is terminated, and control passes to the next statement of the program.

Here is one way UNTIL might be used in the billing program. This version computes PAST_COUNT, but does not make use of INTEREST_FREE:

```
VARIABLES ACCT_NO, OLD_BAL, NEW_BAL, PURCHASE,
   PAST_DUE, INTEREST, PAYMENT ARE INTEGER,
   PAST_COUNT IS INTEGER ::= 0;
UNTIL ACCT_NO <= 0
   REPEAT
      READ_LINE (ACCT_NO, OLD_BAL, PAYMENT, PURCHASE);
      IF PAYMENT >= OLD_BAL THEN
         BEGIN;
            PAST_DUE := 0;
            INTEREST := 0;
         END;
      ELSE
         BEGIN;
            PAST_DUE := OLD_BAL - PAYMENT;
            INTEREST := PAST_DUE * 0.015;
         END;
      NEW_BAL := OLD_BAL - PAYMENT + INTEREST + PURCHASE;
      PRINT (ACCT_NO, OLD_BAL, PAYMENT, PAST_DUE, INTEREST,
         PURCHASE, NEW_BAL);
   END;
PRINT (PAST_COUNT);
```

After each execution of the loop, ACCT_NO <= 0 is evaluated. If ACCT_NO is still positive, the loop continues. If ACCT_NO is not positive, then the REPEAT statement is terminated, and control passes to the next statement -- the statement that prints PAST_COUNT -- which happens to be the last statement. The program then terminates automatically, because the last statement has been executed. (Note that this version of the program produces an extra line of output, because ACCT_NO is not tested until after the PRINT statement in the loop.)

As you can see, a REPEAT qualified with UNTIL does not require a TERMINATE inside the loop. You should be aware that even a REPEAT with an UNTIL phrase can be an infinite loop, if you make a mistake in writing it. It is up to you to insure that the condition tested by UNTIL does always become true after a finite number of loops. There is no way the compiler can detect that you have inadvertently written an infinite loop.

Qualifying REPEAT with the WHILE phrase

A second type of qualifying phrase is the WHILE phrase. Its form is analogous to UNTIL:

```
WHILE condition REPEAT statement-list END
```

Like UNTIL, WHILE tests the BOOLEAN-valued "condition". But there are two notable differences:

1) As its name suggests, WHILE specifies that the REPEAT will be terminated when "condition" is FALSE, and that looping will continue when "condition" is TRUE. This is the opposite of UNTIL's effect.

2) The BOOLEAN expression following WHILE is evaluated before each execution of the loop (whereas the UNTIL condition is evaluated after each execution). In particular, the WHILE condition is evaluated before the first execution of the loop; if it is FALSE at the first evaluation, the REPEAT is terminated before the loop is ever executed. With UNTIL, the REPEAT cannot be terminated until the loop has been executed at least once.

Some loop statements are written more readably and more efficiently with WHILE than with UNTIL. As a simple example, consider a program that reads an INTEGER value N, and if N is positive prints a table of factorials for all integers from 1 to N. (X factorial -- written X! -- is the product of all integers from 1 to X.) This is an implementation using UNTIL:

```
VARIABLE N IS INTEGER,
   MULT, FACT ARE INTEGER ::= 1;
READ_LINE (N);
IF N <= 0 THEN TERMINATE;
UNTIL MULT > N
   REPEAT
      FACT := MULT * FACT;
      PRINT (MULT, FACT);
      MULT := MULT + 1;
   END;
```

If we substitute a WHILE phrase for the UNTIL phrase:

```
WHILE MULT <= N REPEAT ... END;
```

then the IF statement is unnecessary. WHILE will cause MULT <= N to be tested before any execution of the loop. If N <= 0, MULT <= N will evaluate to FALSE, the loop will not be performed at all, and the program will terminate immediately since the REPEAT is its last statement.

Qualifying REPEAT with both WHILE and UNTIL

Occasionally it is useful to qualify a REPEAT with both a WHILE phrase and an UNTIL phrase. The two phrases are written one after the other, in either order:

```
WHILE condition1 UNTIL condition2 REPEAT statement-list END
UNTIL condition2 WHILE condition1 REPEAT statement-list END
```

The WHILE condition is still tested before each loop, and the UNTIL after. The REPEAT is terminated when either test is satisfied -- in other words, when "condition1" is tested and found FALSE, or "condition2" is tested and found TRUE, whichever occurs first.

Both WHILE and UNTIL could be used in the factorial program's REPEAT, to make it stop once FACT exceeds 10000, no matter what the input value of N is:

```
VARIABLE N IS INTEGER,
   MULT, FACT ARE INTEGER ::= 1;
READ_LINE (N);
WHILE MULT <= N UNTIL FACT > 10000
   REPEAT
      FACT := MULT * FACT;
      PRINT (MULT, FACT);
      MULT := MULT + 1;
   END;
```

Only one WHILE or UNTIL phrase may qualify a REPEAT. It would thus be illegal to write:

```
WHILE MULT <= N WHILE FACT <= 10000 REPEAT ... END;
```

The proper way to express two WHILE conditions like this is to connect them with & :

```
WHILE MULT <= N & FACT <= 10000 REPEAT ... END;
```

to produce a single boolean expression that can follow WHILE.

Step-and-Test loops

Let us take a closer look at the problem of printing a table of factorials, which we introduced in the previous sections. This time we will write it with only a simple REPEAT statement:

```
VARIABLE N IS INTEGER,
   MULT, FACT ARE INTEGER ::= 1;
READ_LINE (N);
```

```
IF N <= 0 THEN TERMINATE;
REPEAT
   FACT := MULT * FACT;
   PRINT (MULT, FACT);
   MULT := MULT + 1;
   IF MULT > N THEN TERMINATE;
END;
```

This program employs a step-and-test-loop. Like all loops of this sort, it is identified by three essential features:

1) Before any execution of the loop, MULT is initialized to 1.

2) At the end of each pass through the loop, MULT is stepped by 1.

3) Immediately after stepping, MULT is tested against a limit, N. If MULT exceeds N, the loop is terminated; if not, the loop is executed again.

Step-and-test loops are extremely common in computer programming. So CS-4 provides four qualification phrases which are designed to make step-and-test loops easier to write. The phrases may be used together, like this:

```
FOR variable FROM expression BY expression THRU expression
   REPEAT statement-list END
```

where "variable" is the name of a variable (usually an INTEGER) and "expression" is an arithmetic expression. The meanings of the phrases are as follows:

FOR phrase -- specifies a variable (sometimes referred to as the "control variable") which is to be initialized before looping, and stepped and tested after each loop.

FROM phrase -- specifies the initial value.

THRU phrase -- specifies the limit value against which each test is made.

BY phrase -- specifies a step value.

Using these qualifying phrases, the factorial program can be written more compactly and more readably:

```
VARIABLES N, MULT ARE INTEGER,
   FACT IS INTEGER ::= 1;
READ_LINE (N);
IF N <= 0 THEN TERMINATE;
FOR MULT FROM 1 THRU N BY 1
   REPEAT
      FACT := MULT * FACT;
      PRINT (MULT, FACT);
   END;
```

The statement-list following REPEAT will be executed exactly N times, and N lines will be printed out.

It is not necessary to write the individual qualifying phrases in any particular order. The REPEAT statement above could also have been written:

```
FOR MULT FROM 1 BY 1 THRU N REPEAT ... END;
FROM 1 FOR MULT THRU N BY 1 REPEAT ... END;
THRU N BY 1 FROM 1 FOR MULT REPEAT ... END;
BY 1 FOR MULT THRU N FROM 1 REPEAT ... END;
```

or any of 19 other possible ways. But it is a good idea to choose one easy-to-read order, and stick with it in all your programs.

Using the step-and-test qualifiers in special cases

Let us take as an example an even simpler program than the one we used in the previous section. This one moves the PRINT statement outside the loop, so that it prints only N!:

```
VARIABLES N, MULT ARE INTEGER,
   FACT IS INTEGER ::= 1;
READ_LINE (N);
IF N <= 0 THEN TERMINATE;
FOR MULT FROM 1 THRU N BY 1 REPEAT FACT := MULT * FACT; END;
PRINT (N, FACT);
```

(Note that the PRINT statement now prints N instead of MULT; alternatively, we could have written

```
PRINT (MULT, FACT);
```

since MULT is always equal to N when the REPEAT is terminated.)

The IF statement in this program insures that N is always at least 1 when the REPEAT statement is executed. So the THRU value is never less than the FROM value, and the loop is always executed at least once.

What would happen if the IF were left out? In that case, the THRU value might be less than the FROM value. In other words, the initial value of MULT might already be greater than the limit. When this happens, the effect is the same as when a WHILE condition is initially FALSE: the REPEAT statement is terminated immediately, and the loop is never executed.

This behavior of REPEAT can often be used to advantage. For example, we can make the sample program more efficient by setting the initial value of MULT at 2:

```
FOR MULT FROM 2 THRU N BY 1 REPEAT FACT := MULT * FACT; END;
```

When N is 2 or greater, the loop is executed N - 1 times; the superfluous step of calculating MULT * FACT when MULT is 1 is eliminated. When N is 1, the THRU value is less than the FROM value, so the loop is not executed at all. The value of FACT remains equal to 1, which is what it should be when N is 1.

There is one other situation when the loop is never executed: when the BY value is equal to zero. This makes it impossible to accidentally write an infinite loop that steps a variable by zero forever. It can also occasionally be used to advantage in other cases.

Negative step values

It is permissible to use a negative BY value. However, it causes the REPEAT statement to act differently in some respects:

1) The FOR variable is stepped down after each execution of the loop.

2) Looping continues until the FOR variable is stepped below the THRU value.

3) The loop statement is never executed if the THRU value is greater than the FROM value.

As an example, we could have replaced the REPEAT statement in the program in the previous section by

```
FOR MULT FROM N THRU 2 BY -1 REPEAT FACT := MULT * FACT; END;
```

The result is the same, for any N.

REAL mode in REPEAT statements

So far all our examples have used INTEGER values and variables in the qualifying phrases. However, all the rules are the same if you use REAL values and variables.

If a FROM, BY, or THRU value is not of the same mode as the FOR variable, the value is converted to the mode of the variable. The conversion is performed before the regular actions of the REPEAT statement. In particular, if a FROM, BY, or THRU value is of REAL mode, and the FOR variable is INTEGER, the value will be rounded before the REPEAT statement is carried out.

As an example, take

```
VARIABLE I IS INTEGER;
VARIABLE R IS REAL;
FOR R FROM 1.7 THRU 4.1 BY 0.6 REPEAT ... END;
FOR I FROM 1.7 THRU 4.1 BY 0.6 REPEAT ... END;
```

The first REPEAT statement initializes R to 1.7, and steps it by 0.6 until its value exceeds 4.1. The loop will be executed five times in all. In the second REPEAT, the values in the qualifying phrases are rounded first. So it is executed the same as if it read

```
FOR I FROM 2 THRU 4 BY 1 REPEAT ... END;
```

The loop will be executed only three times.

In any step-and-test loop, the FOR variable is involved in a comparison after each execution of the loop. It is therefore best to avoid REAL-mode FOR variables whenever possible, because of the inexactness in REAL comparison that we mentioned in Chapter 5. If it is necessary to use REAL FOR variables, it is a good idea to arrange the THRU value so that the last test is sure to produce the desired result. For example, to step from 1.7 through 4.1 by 0.6, you could write

```
FOR R FROM 1.7 THRU 4.4 BY 0.6 REPEAT ... END;
```

Then when R is stepped to 4.1, it is distinctly less than 4.4, and execution of the loop continues; the next value for R is 4.7, which clearly fails the test and terminates the REPEAT.

Default actions of step-and-test qualifying phrases

In many cases you can shorten the REPEAT statement by leaving out one or more qualifying phrases. Whenever a step-and-test qualifying phrase is left out, the compiler assumes that a standard default action was intended. (The term "default" refers to something the compiler does for you when you leave out information it needs to know. It does not mean that you made an error.)

When FROM or BY phrases are omitted, default values of 1 are assumed. Thus the loop in the factorial program:

```
FOR MULT FROM 1 THRU N BY 1 REPEAT FACT := MULT * FACT; END;
```

can be written with any of these shorter forms:

```
FOR MULT FROM 1 THRU N REPEAT FACT := MULT * FACT; END;
FOR MULT THRU N BY 1 REPEAT FACT := MULT * FACT; END;
FOR MULT THRU N REPEAT FACT := MULT * FACT; END;
```

If the variable in a FOR phrase is not actually used in the loop, then the FOR phrase can be omitted. The compiler assumes that a REAL-mode FOR variable was intended. For instance, here is a loop which sets POWER to M^N:

```
VARIABLES MULT, M, N ARE INTEGER,
   POWER IS INTEGER ::= 1;
READ_LINE (M, N);
FOR MULT THRU N REPEAT POWER := POWER * M; END;
```

Here MULT does not appear in the loop at all, so the FOR phrase can be dropped:

```
THRU N REPEAT POWER := POWER * M; END;
```

(Of course, the compiler assumes a REAL FOR variable, whereas we originally used an INTEGER one; but in this case -- and many others -- it makes no difference.)

When the THRU phrase is omitted, the FOR variable is stepped without limit. To avoid creating an infinite loop, some other means must be provided to terminate the REPEAT statement.

One use of a REPEAT without a THRU phrase is to count the number of times through the loop. For example, our billing program makes one loop for each account processed, so we can use a REPEAT like this to count the number of processed accounts:

```
VARIABLES ACCT_NO, OLD_BAL, NEW_BAL, PURCHASE, PAST_DUE,
   INTEREST, PAYMENT, COUNT ARE INTEGER;
FOR COUNT FROM 0
   REPEAT
      READ_LINE (ACCT_NO, OLD_BAL, PAYMENT, PURCHASE);
      IF ACCT_NO <= 0 THEN
         BEGIN;
            PRINT (COUNT);
            TERMINATE;
         END;
      . . .
   END;
```

COUNT will be stepped until some input value of ACCT_NO is non-positive. Then COUNT is printed out before the program terminates.

Using both step-and-test and comparison qualifying phrases

It is legal, and often useful, to combine WHILE or UNTIL or both with FOR, FROM, THRU, and BY. The loop may then be terminated by a false WHILE condition, a true UNTIL condition, or the stepping of the FOR variable past its limit, whichever occurs first. The rules are:

1) A WHILE condition is always tested immediately before each loop begins. It is thus tested after initialization the first time, and after stepping each subsequent time.

2) An UNTIL condition is always tested immediately after each loop ends. It is therefore tested before stepping.

Here is a simple example which combines a step-and-test loop with WHILE. It prints out a table of numbers and their factorials:

```
VARIABLES N, MULT ARE INTEGER,
   FACT IS INTEGER ::= 1;
READ_LINE (N);
FOR MULT THRU 20 WHILE FACT <= N
   REPEAT
      FACT := MULT * FACT;
      PRINT (MULT, FACT);
   END;
```

The last line in the table will be for 20, or the greatest number whose factorial is less than N -- whichever is smaller. Note that if N is less than 1, nothing is printed out.

WHILE or UNTIL can also be used to insure termination of a REPEAT statement that has a FOR phrase but no THRU phrase. For instance, you could also write the billing program example of the previous section as

```
FOR COUNT FROM 0 UNTIL ACCT_NO <= 0
   REPEAT
      . . .
   END;
PRINT (COUNT);
```

Nested REPEAT statements

REPEAT statements used in combination can produce very compact but powerful programs -- especially if one REPEAT is nested inside the loop of the other. Then the inner REPEAT will itself be executed repeatedly.

As an example, we present here a program to compute all prime numbers less than or equal to some input value N. A prime is a number that can't be evenly divided by any number between 1 and itself -- such as 2, 3, 5, 7, 11, or 13 (to name the first six). Two is the only even prime, since all other even numbers are divisible by 2. There is an infinite number of odd primes.

To check if an odd number is prime, we try dividing it by odd numbers less than it, starting with 3. We can save some time by taking advantage of the fact that a non-prime number's divisors come in pairs; for instance we can break down 225 as:

```
225 =  3 * 75
    =  5 * 45
    =  9 * 25
    = 15 * 15
```

For any number N, each divisor greater than the square root of N (15 in our example) has a corresponding divisor less than the square root of N. So if we don't find any divisors less than the square root of N, we can assume N is prime.

```
VARIABLES N, CANDIDATE, TEMP, DIVISOR ARE INTEGER,
   EVEN_DIVIDE IS BOOLEAN ::= FALSE;
READ_LINE (N);
IF N >= 2 THEN PRINT (2);
ELSE TERMINATE;
FOR CANDIDATE FROM 3 THRU N BY 2
   REPEAT
      FOR DIVISOR FROM 3 THRU CANDIDATE ** 0.5 BY 2
         UNTIL EVEN_DIVIDE
         REPEAT
            TEMP := CANDIDATE / DIVISOR;
            EVEN_DIVIDE := DIVISOR * TEMP = CANDIDATE;
         END;
      IF ~EVEN_DIVIDE THEN PRINT (CANDIDATE);
   END;
```

The best way to appreciate the nuances of this program is to follow it through for a few small values of N. The general logic of the program is as follows:

CANDIDATE ranges through all odd numbers from 3 to N. Each pass through the outer loop tests whether some value of CANDIDATE is prime.

For each value of CANDIDATE, DIVISOR can range through all odd numbers from 3 to CANDIDATE ** 0.5 (the square root of CANDIDATE). Each pass through the inner loop checks some value of DIVISOR to see if it evenly divides CANDIDATE. EVEN_DIVIDE is set to TRUE or FALSE accordingly.

The inner loop is terminated as soon as EVEN_DIVIDE is TRUE for some value of DIVISOR. If no value of DIVISOR evenly divides CANDIDATE, then the loop terminates when DIVISOR steps past the limit, and EVEN_DIVIDE remains FALSE.

If the inner loop terminates with EVEN_DIVIDE = FALSE, then CANDIDATE is prime, so it is printed. If the inner loop terminates with EVEN_DIVIDE = TRUE, then CANDIDATE is not a prime and is not printed.

Leaving out the looped statements

Some simple tasks can be accomplished by a REPEAT that has no looped statement at all. An example is:

```
FOR I WHILE 2 ** I - 1 < TEST_VALUE REPEAT; END;
```

This statement finds the smallest positive I such that 2^I-1 is greater than or equal to TEST_VALUE. There is no looped statement to be executed; so after each test of the WHILE condition, the REPEAT goes on immediately to step and test I again.

There are two ways to look at this situation. You can think of this as a special form of the REPEAT statement, in which no looped statement follows REPEAT -- and so there is no execution of a looped statement. Or you can imagine that the compiler automatically inserts an empty statement when it finds no statement between the words REPEAT and END. An empty statement can be thought of as a space filler, which has no effect on anything when it is executed; it's just put there to follow the rules, because the rules specify a statement-list between REPEAT and END.

The empty statement approach is broader -- it doesn't require a special form to describe cases where a statement can be left out. There are actually many such cases in CS-4. In fact, we can make a quite general rule: anywhere other CS-4 executable statements may appear, an empty statement may be left instead. An empty statement can thus come, for instance, after THEN:

```
IF N = 0 THEN;
ELSE N := N + 1;
```

in which case no action is performed when the THEN clause is executed -- control just skips over ELSE to the next statement.

9.0
MORE ABOUT BLOCKS AND LOOPS: THE EXIT STATEMENT

When we introduced BEGIN blocks and REPEAT loops we said that they could be terminated by passing control to END, or by terminating the whole program through TERMINATE. There is one more very useful way to terminate a block or loop in CS-4 -- by executing an EXIT statement.

EXIT can be used in a number of powerful ways. A discussion of some of these uses will have to wait until you are more proficient in the language. But at this point we can introduce two of the simpler forms of EXIT, and show how to use EXIT to simplify the writing of REPEAT loops.

Using EXIT

The most elementary form of the EXIT statement is written simply:

```
EXIT;
```

When this statement is executed, it causes control to pass directly to the point following END. A BEGIN block is thus terminated immediately; any statements between the EXIT and END are skipped. As a simple example, we could write

```
BEGIN;
   PAST_DUE := OLD_BAL - PAYMENT;
   IF PAYMENT <= OLD_BAL THEN EXIT;
   PAST_DUE := 0;
END;
INTEREST := PAST_DUE * 0.015;
```

Suppose this block is executed with PAYMENT <= OLD_BAL. Then the IF statement causes EXIT to be executed, terminating the block immediately. Control passes directly to the next statement following the block -- the assignment to INTEREST. That assignment is the next statement that is executed. The assignment PAST_DUE := 0 is not executed at all. On the other hand, if PAYMENT is greater than OLD_BAL, the IF does not execute EXIT. The block is eventually concluded when control reaches END in the normal sequence -- after PAST_DUE is assigned 0.

EXIT is meaningful only when it appears within an executable block, such as the BEGIN block above, or a REPEAT loop statement list. If you write EXIT where it is not meaningful, the compiler will signal an error and will not let the program be executed.

We have said that a BEGIN block is executed as if it were a sort of "sub-program" in itself. You can think of EXIT as having the same relation to a BEGIN block "sub-program" that TERMINATE has to whole programs. Both cause immediate termination -- but TERMINATE terminates an entire program, whereas EXIT just terminates the "sub-program" which is contained in the block.

REPEAT loops containing EXIT

REPEAT loops, like BEGIN blocks, contain a list of statements followed by END. Each execution of the statement-list is one pass through the loop; a pass through the loop ends when control reaches the END of the list. When one of the conditions in the REPEAT loop's qualifying-phrase-list is met, the REPEAT loop stops looping and control passes to the point following END. The execution of EXIT in a REPEAT loop, as in a BEGIN block, causes control to pass immediately to the point following END. So when EXIT is executed in a REPEAT loop, the looping process terminates, just as it would if one of the conditions in the qualifying-phrase-list were suddenly tested and met. EXIT, though, has a versatility that the qualifying-phrase-list doesn't have: statements containing EXIT may appear anywhere within the loop. EXIT enables you to test for a condition that is not specified in the qualifying-phrase-list, and test it at whatever point in the statement list you choose to place the statement containing EXIT.

There is another important way in which EXIT statements may be used in REPEAT loops. Sometimes you want to exit not from the entire looping process, but from just one pass through the loop. You want to make the control pass directly to END, skipping over the intervening statements. Then you want to have the conditions of the REPEAT loop tested (as they would if control had reached END normally) to determine whether the loop should be executed again. This effect may be achieved using EXIT, if you simply place the entire statement-list portion of the REPEAT statement within a BEGIN block. An EXIT from such a BEGIN block has the effect of passing control immediately to the "bottom" of one pass through the loop.

There are many natural applications of EXITing from one pass through a REPEAT loop. For instance, you might want the billing program to skip over accounts that did not change for the month -- accounts with OLD BAL, PAYMENT, and PURCHASE all zero. If you place the entire statement list within a BEGIN block, you then can put in an IF with an EXIT statement to do the job:

```
VARIABLES ACCT_NO, OLD_BAL, NEW_BAL, PURCHASE, PAST_DUE,
   INTEREST, PAYMENT, PAST_COUNT ARE INTEGER ::= 0;
UNTIL ACCT NO <= 0
   REPEAT
      BEGIN;
         READ_LINE (ACCT_NO, OLD_BAL, PAYMENT, PURCHASE);
         IF OLD_BAL = 0 & PAYMENT = 0 & PURCHASE = 0 THEN EXIT;
         IF PAYMENT >= OLD_BAL THEN
            BEGIN;
               PAST_DUE := 0;
               INTEREST := 0;
            END;
         ELSE
            BEGIN;
               PAST_DUE := OLD_BAL - PAYMENT;
               INTEREST := PAST_DUE * 0.015;
               PAST_COUNT := PAST_COUNT + 1;
            END;
         NEW_BAL := OLD_BAL - PAYMENT + INTEREST + PURCHASE;
         PRINT (ACCT_NO, OLD_BAL, PAYMENT, PAST_DUE,
            INTEREST, PURCHASE, NEW_BAL);
      END;
   END;
PRINT (PAST_COUNT);
```

Notice that there are <u>two</u> END statements just before PRINT. One of these marks the end of the BEGIN block; the other marks the end of the REPEAT loop. When the dollar input values are all zero, EXIT gets executed. It terminates the BEGIN block and that pass through the loop -- the rest of the statements in it are skipped over. Then the UNTIL phrase is tested, and if it yields FALSE the loop starts again at the beginning of the BEGIN block. On the other hand, if EXIT is not executed the loop ends in the usual way, when control passes to END after the statement that prints ACCT_NO and all the dollar values.

Note that with the entire loop within a BEGIN block, EXIT does not cause the REPEAT statement to stop looping. It just causes one particular pass through the loop to end, and another pass to begin immediately after the test. The only way looping can be stopped is for ACCT_NO to be non-positive; only then will PRINT (PAST_COUNT) finally be executed.

The need to label blocks

Suppose you tried to make a slight modification to the program we just wrote, so that ACCT_NO was still printed out when an account required no processing. You might want to simply add PRINT (ACCT_NO) to the first IF statement, just before the EXIT:

```
UNTIL ACCT_NO <= 0
   REPEAT
      BEGIN;
         READ_LINE (ACCT_NO, OLD_BAL, PAYMENT, PURCHASE);
         IF OLD_BAL = 0 & PAYMENT = 0 & PURCHASE =  0 THEN
            BEGIN;
              PRINT (ACCT_NO);
              EXIT;
            END;
         . . .
      END;
   END;
```

But this does not do what you want it to. The problem is that now EXIT is inside two BEGIN blocks -- an outer one, (enclosing all the statements within the REPEAT loop), and an inner one (the THEN clause). The EXIT statement does not say which block is to be terminated, so the compiler assumes you want to terminate only the innermost block. The outer block -- the one you wanted to terminate -- still continues normally until END is reached. The same problem arises when EXIT is placed in a REPEAT statement that is within the block you wish to EXIT from.

To resolve this problem, CS-4 lets you specify explicitly which block you want to terminate. But to do this, you first have to put a label on the block in question, so that you have a way to refer to it. A label is a name followed by a colon. For a BEGIN block it is placed just before the word BEGIN, usually on a separate line so it stands out:

```
NEW_INPUT_CASE:
   BEGIN;
      . . .
```

The label name is also placed after the word END in the labelled block's END statement:

```
NEW_INPUT_CASE:
   BEGIN;
      . . .
   END NEW_INPUT_CASE;
```

Actually, labelling the END is optional -- the compiler accepts unlabelled ENDs on labelled blocks. Labelled ENDs are preferable, however, because they improve the program's readability.

REPEAT statements may be labelled in the same way as BEGIN blocks. EXIT and EXIT FROM behave the same for REPEAT statements as they do for BEGIN blocks. (So for simplicity's sake, in the rest of this chapter we will use the term "block" to mean either a BEGIN block or a REPEAT statement.)

Labelling a block does not change the way it is executed. It just gives the block a name, so you can refer to it. The rules for forming label names are the same as for variable names. A name may not be used for both a label and a variable at the same time.

Referring to a labelled block in an EXIT statement

To terminate a block by name, you add a FROM phrase to the EXIT statement:

```
EXIT FROM label;
```

where "label" is the label name of one of the blocks that contains the EXIT statement. The named block -- and all blocks nested within it -- will be terminated immediately.

Now we can get around the problem that came up in the previous section, by labelling the outer block:

```
UNTIL ACCT_NO <= 0
   REPEAT
   NEW_ACCT:
      BEGIN;
         READ_LINE (ACCT_NO, OLD_BAL, PAYMENT, PURCHASE);
         IF OLD_BAL = 0 & PAYMENT = 0 & PURCHASE = 0 THEN
            BEGIN;
               PRINT (ACCT_NO);
               EXIT FROM NEW_ACCT;
            END;
         . . .
       END NEW_ACCT;
    END;
```

The EXIT statement specifies that the outer block as well as the inner one is to be terminated.

The rules for EXIT can be summarized like this: an EXIT statement transfers control to the point following the end of the block named in its FROM phrase. In the process, any blocks nested within that named block which are also being executed are also terminated. If there is no FROM phrase, only the innermost block is terminated. (If none of the blocks containing the EXIT statement is labelled with the name in the FROM phrase, then the compiler signals an error, and the program cannot be executed.)

EXITing from a nested REPEAT

An EXIT statement with a FROM phrase can be used to simplify the prime-generating program we presented at the end of the last chapter, so that there is no need to set or test the BOOLEAN variable EVEN_DIVIDE.

```
VARIABLES N, CANDIDATE, TEMP, DIVISOR ARE INTEGER;
READ_LINE (N);
IF N >= 2 THEN PRINT (2);
ELSE TERMINATE;
FOR CANDIDATE FROM 3 THRU N BY 2
   REPEAT
   TEST:
      BEGIN;
         FOR DIVISOR FROM 3 THRU CANDIDATE ** 0.5 BY 2
            REPEAT
               TEMP := CANDIDATE / DIVISOR;
               IF DIVISOR * TEMP = CANDIDATE THEN EXIT FROM TEST;
            END;
         PRINT (CANDIDATE);
      END TEST:
   END;
```

If some value of DIVISOR is found to evenly divide CANDIDATE, then

```
EXIT FROM TEST;
```

is executed. It causes control to pass directly to the point following END TEST, which is the END of the outer loop. As a result, the inner

REPEAT statement is terminated immediately: DIVISOR is not stepped or tested any further, and no more passes are made through the inner loop. CANDIDATE, the outer loop's FOR variable, is immediately stepped by 2, and, if it does not exceed N, the outer loop starts once more at the beginning. (One thing this example demonstrates is that an EXIT can terminate a REPEAT statement, if it terminates a block containing the REPEAT.)

Of course, the EXIT statement also causes PRINT (CANDIDATE) to be skipped over. So when CANDIDATE is divisable by some DIVISOR value, it is not printed out as a prime. On the other hand, when CANDIDATE is prime the EXIT never gets executed, and the inner REPEAT terminates when divisor is stepped past CANDIDATE ** 0.5. Then CANDIDATE does get printed out.

PART 2

DATA HANDLING WITH ARRAYS

AND CHARACTER STRINGS

10.0

DECLARING AND USING ARRAYS OF VALUES

Very often, computer programs deal with sets of values. Some examples are:

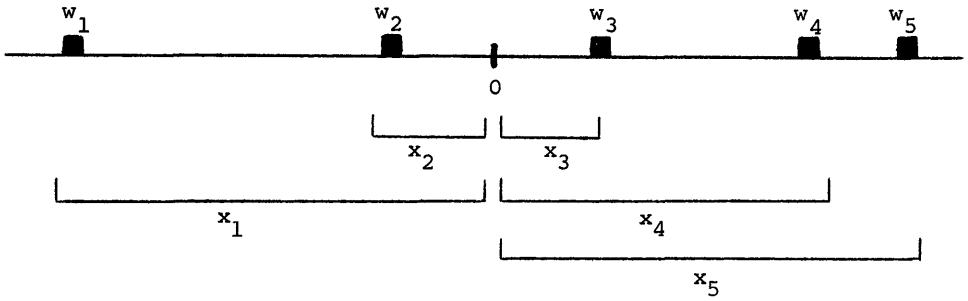
inputs from a line or field of sensors
a list of primes
vectors
scores on a series of tests
a list of invalid account numbers

Programs would become quite cumbersome if you had to declare a separate variable name to refer to each value in a set. So CS-4 lets you declare array variables, which refer to whole collections of values.

This chapter is an introduction to the use of CS-4 arrays. It explains how to write declarations for them, how to refer to individual values in an array, and how to input and output arrays; and it presents some simple programs using arrays of INTEGER and REAL values.

The need for arrays

Let's consider a simple physics problem, which will show how useful arrays can be. We want to write a program to find the center of gravity of a rod on which five weights are hung. We assume the weight of the rod is negligible. The situation can be drawn like this:



The w_i are the individual weights and the x_i are the corresponding positions relative to the center point, 0. (Positions are measured positive to the right, and negative to the left.) The formula for the position of the center of gravity (again in terms of distance from the center point) is:

$$C = (w_1x_1+w_2x_2+w_3x_3+w_4x_4+w_5x_5)/(w_1+w_2+w_3+w_4+w_5)$$

or, using the more compact Σ-notation for sums:

$$C = \sum_{i=1}^{5} w_i x_i \; / \; \sum_{i=1}^{5} w_i$$

The input for each case will be on two sequences of cards -- the weights first, and then the corresponding distances.

It is possible to write a program for this problem without using array variables. The program would have to declare five variables for weights, and five for distances, and would look something like this:

```
VARIABLES WEIGHT1, WEIGHT2, WEIGHT3, WEIGHT4, WEIGHT5,
   DIST1, DIST2, DIST3, DIST4, DIST5, WEIGHT_SUM,
   CENTER ARE REAL;
REPEAT
   READ_LINE (WEIGHT1, WEIGHT2, WEIGHT3, WEIGHT4, WEIGHT5,
      DIST1, DIST2, DIST3, DIST4, DIST5);
   WEIGHT_SUM := WEIGHT1 + WEIGHT2 + WEIGHT3 + WEIGHT4 +
      WEIGHT5;
   IF WEIGHT_SUM = 0 THEN TERMINATE;
   CENTER := (WEIGHT1*DIST1 + WEIGHT2*DIST2 + WEIGHT3*DIST3 +
      WEIGHT4*DIST4 + WEIGHT5*DIST5) / WEIGHT_SUM;
   PRINT (WEIGHT1, WEIGHT2, WEIGHT3, WEIGHT4, WEIGHT5,
      DIST1, DIST2, DIST3, DIST4, DIST5, CENTER);
END;
```

The IF statement serves two purposes. First, it prevents division by zero in the calculation of CENTER. (If a program does try to divide by zero, its execution is immediately terminated, and an error message is printed in the output.) Second, the IF provides a convenient way to terminate the program correctly -- just set all five weights equal to zero on the last input card.

There are several reasons why you would want to improve on this program. It is somewhat bulky and hard to read. It is useful only for problems with five weights -- if you wanted to have, say, six input weights and distances, you would have to declare two more variables

(called, perhaps, WEIGHT6 and DIST6), and practically every statement would have to be changed. The more inputs you want to have, the bulkier you must make the program. To handle 25 input weights, you would have to declare 50 different variables for input, and the expression to compute CENTER would have 25 additions and 25 multiplications written out.

Finally, there is no simple way to adapt the above program to handle a variable number of inputs. Each possible number of inputs requires a different set of statements. For example, suppose for each case you first read a value N that tells how many weights there are for that case; suppose N can be 3, 4, or 5. Just to read in the input values, you would have to write:

```
READ_LINE (N);
IF N = 3 THEN
   READ_LINE (WEIGHT1, WEIGHT2, WEIGHT3, DIST1,
      DIST2, DIST3);
ELSE
   IF N = 4 THEN
      READ_LINE (WEIGHT1, WEIGHT2, WEIGHT3, WEIGHT4,
         DIST1, DIST2, DIST3, DIST4);
   ELSE
      IF N = 5 THEN
         READ_LINE (WEIGHT1, WEIGHT2, WEIGHT3, WEIGHT4,
            WEIGHT5, DIST1, DIST2, DIST3, DIST4, DIST5);
      ELSE TERMINATE;
```

and the rest of the program would be more complicated, too (try it and see). A program to take any number of weights from 1 to 25 would fill many pages.

Declaring array variables

A concise and flexible program for the center-of-gravity problem can be written by declaring a variable to represent more than one value of a given mode. A variable so declared is called an array variable. Like all variables, it must be named in a declaration statement.

As an example, suppose that instead of declaring five individual weight variables

```
VARIABLES WEIGHT1, WEIGHT2, WEIGHT3, WEIGHT4,
   WEIGHT5 ARE REAL;
```

you want to declare a single variable WEIGHT that has five REAL values. You would write a declaration for WEIGHT like this:

```
VARIABLE WEIGHT IS ARRAY(5) REAL;
```

The word ARRAY coming directly after IS (or ARE) indicates that WEIGHT is an array variable. The number in parentheses after ARRAY tells the number of elements in WEIGHT. Then comes the mode name -- REAL -- that indicates what mode each element of the array is.

You can declare arrays of any number of elements of any mode; for example:

```
VARIABLE SCORE IS ARRAY(100) INTEGER;
VARIABLES TEST1, TEST2 ARE ARRAY(2) BOOLEAN;
```

SCORE is here declared to represent an array of 100 elements of INTEGER mode. TEST1 and TEST2 represent arrays of two BOOLEAN elements.

ARRAY is our first example of what is called a mode generator. It has the effect of generating new modes from old ones. You can think of the expression

```
ARRAY(5) REAL
```

as the name of a new mode. The "values" that this new mode represents are lists of five numerical values -- whereas the values that the mode REAL represents are single numerical values. When you write

```
VARIABLE WEIGHT IS ARRAY(5) REAL;
```

you can think of it as saying that the mode of WEIGHT is ARRAY(5) REAL.

Referring to individual array elements

The name of an array variable can be very useful all by itself. Nonetheless, it is necessary to have a way of naming each element of the array, so that individual elements can have operations performed on them, and can be read or printed.

The way array elements are referred to in CS-4 has a lot in common with familar mathematical notation. Remember that when we defined the center of gravity problem, we used the letter w to stand for weights. The individual weights were represented by subscripted w's:

w_1, w_2, w_3, w_4, w_5

The sum of the weights was written

$$\sum_{i=1}^{5} w_i$$

where w_i represented each of the weights w_1 through w_5, as i ranged from 1 through 5.

In CS-4, we also write subscripts to select individual values. However, there is no way to write a number "below the line" on a punched card. Subscripts are indicated in a different way -- by putting them in parentheses after the array name. So, given

```
VARIABLE WEIGHT IS ARRAY(5) REAL;
```

the five elements of WEIGHT are written

```
WEIGHT(1), WEIGHT(2), WEIGHT(3), WEIGHT(4), WEIGHT(5)
```

Furthermore, if you declare an individual variable

```
VARIABLE I IS INTEGER;
```

then you are allowed to write

```
WEIGHT(I)
```

which can represent any of the values WEIGHT(1) through WEIGHT(5), depending on the current value of I. If WEIGHT(I) appears in a loop, and the value of I is different each time through the loop, then WEIGHT(I) will represent a different element of WEIGHT each time through the loop. So you can calculate the sum of all the values in array WEIGHT like this:

```
VARIABLE WEIGHT_SUM IS INTEGER ::= 0;
FOR I THRU 5 REPEAT WEIGHT_SUM := WEIGHT_SUM + WEIGHT(I); END;
```

Expressions such as WEIGHT(1) and WEIGHT(I) are subscripted variable names. They represent single values in the array WEIGHT. In the present example they represent REAL values, since WEIGHT is an array of REALs. They can be used in a program in all the same ways that non-subscripted REAL-mode variables can be used. For example:

`WEIGHT(3) := 0;`	assign the value 0 to the third element of WEIGHT
`IF WEIGHT(1) > WEIGHT(2) THEN TERMINATE;`	execute TERMINATE if the first element of WEIGHT has a greater value than the second element
`WEIGHT(I) := WEIGHT(I) + 1;`	the Ith element of WEIGHT is assigned a new value equal to its present value plus 1
`IF TEST(3) THEN TERMINATE;`	TEST must be declared as an array of BOOLEAN elements; the statement causes TERMINATE to be executed if the third element of TEST has the value TRUE

Subscripts are not limited to literals and single variables. Any arithmetic expression may be used as a subscript. For instance, you can legally write

```
WEIGHT(2 * I - 1)
```

which refers to WEIGHT(1) when I is 1, to WEIGHT(3) when I is 2, and to WEIGHT(5) when I is 3.

If you use variables in a subscript expression, you must be careful that the subscript value remains in bounds. If I is 4, reference to WEIGHT(2 * I - 1) is in error, since there is no element WEIGHT(7).

Only expressions that yield INTEGER values may be used as subscripts.

Reading and printing arrayed values

To assign input values to elements of an array, you can put subscripted variables in a READ_LINE statement. For instance, here we read values into all five elements of array WEIGHT:

```
VARIABLE WEIGHT IS ARRAY(5) REAL;
READ_LINE (WEIGHT(1), WEIGHT(2), WEIGHT(3), WEIGHT(4),
   WEIGHT(5));
```

You can also do the same thing, more compactly, by writing just the array name in the READ_LINE statement:

```
READ_LINE (WEIGHT);
```

This statement causes one value to be read for each element of WEIGHT. The first value read is assigned to WEIGHT(1), the second to WEIGHT(2), and so on. Of course, if you only want to read values into certain elements, you have to specify them explicitly with subscripted variables.

The operation of the PRINT statement is similar. To print out all the elements of WEIGHT in order, you write:

```
PRINT (WEIGHT);
```

To write out just the first, fifth, and third elements, in that order, you write

```
PRINT (WEIGHT(1), WEIGHT(5), WEIGHT(3));
```

Writing a program using arrays

Now we are prepared to rewrite the center of gravity program in an improved form. The input values are read into two arrays of five elements -- called WEIGHT and DISTANCE -- and then the elements of these arrays are multiplied and summed in a loop.

```
VARIABLES WEIGHT, DISTANCE ARE ARRAY(5) REAL,
   WEIGHT_SUM, PRODUCT_SUM, CENTER ARE REAL,
   I IS INTEGER;
```

```
REPEAT
   WEIGHT_SUM := 0;
   PRODUCT_SUM := 0;
   READ_LINE (WEIGHT, DISTANCE);
   FOR I THRU 5
      REPEAT
        WEIGHT_SUM := WEIGHT_SUM + WEIGHT(I);
        PRODUCT_SUM :=
          PRODUCT_SUM + WEIGHT(I) * DISTANCE(I);
      END;
   IF WEIGHT_SUM = 0 THEN TERMINATE;
   CENTER := PRODUCT_SUM / WEIGHT_SUM;
   PRINT (WEIGHT, DISTANCE, CENTER);
END;
```

When the inner loop is terminated, WEIGHT_SUM is the sum of all the weights, and PRODUCT_SUM is the sum of the weight-distance products. If WEIGHT_SUM is not zero, CENTER is computed as PRODUCT_SUM / WEIGHT_SUM.

Only two slight changes need be made to this program to process any other number of input weights. To handle, say, 25 weights, the changed lines would be

```
VARIABLES WEIGHT, DISTANCE ARE ARRAY(25) REAL;
. . .
FOR I THRU 25
```

The program stays the same size, no matter how many inputs it is written to handle.

Array inputs of varying size

Once you declare that an array has a fixed number of elements, you cannot change its size during execution. If you write

```
VARIABLES WEIGHT, DISTANCE ARE ARRAY(25) REAL;
```

then WEIGHT and DISTANCE have 25 elements throughout the program. As a result, every time the statement

```
READ_LINE (WEIGHT, DISTANCE);
```

is executed, 25 values are read into the array WEIGHT, and 25 into DISTANCE.

Even so, there are ways to write a program to accept a varying number of input values. One method that works for the center of gravity problem is to precede each set of WEIGHT and DISTANCE input cards with a separate "size" input card. The size card contains an INTEGER value that indicates the number of WEIGHT and DISTANCE values which follow.

To implement this arrangement, we add a READ_LINE statement at the beginning of the program loop which reads the size value into a variable N. Input values are then read into elements 1 through N of WEIGHT and DISTANCE, and the loop which calculates WEIGHT_SUM and PRODUCT_SUM is performed with I ranging from 1 to N.

```
VARIABLES WEIGHT, DISTANCE ARE ARRAY(25) REAL,
   WEIGHT_SUM, PRODUCT_SUM, CENTER ARE REAL,
   I, N ARE INTEGER;
REPEAT
   WEIGHT_SUM := 0;
   PRODUCT_SUM := 0;
   READ_LINE (N);
   IF N < 1 | N > 25 THEN TERMINATE;
   FOR I THRU N REPEAT READ_LINE (WEIGHT(I)); END;
   FOR I THRU N REPEAT READ_LINE (DISTANCE(I)); END;
   FOR I THRU N
      REPEAT
         WEIGHT_SUM := WEIGHT_SUM + WEIGHT(I);
         PRODUCT_SUM :=
            PRODUCT_SUM + WEIGHT(I) * DISTANCE(I);
      END;
   IF WEIGHT_SUM = 0 THEN TERMINATE;
   CENTER := PRODUCT_SUM / WEIGHT_SUM;
   FOR I THRU N REPEAT PRINT (WEIGHT(I)); END;
   FOR I THRU N REPEAT PRINT (DISTANCE(I)); END;
   PRINT (CENTER);
END;
```

This program handles any number of inputs from 1 to 25. It could handle a greater number of inputs if the sizes of the arrays were increased. (Note the necessity of an additional IF statement, to check that the value of N is in bounds.)

There is one defect in the program as it now stands. The statement READ_LINE (WEIGHT(I)) is executed N times for each case. Each time it is executed, it starts reading a new card -- so each input WEIGHT value must be on a separate card. The same goes for each DISTANCE value. And we have a similar problem with the PRINT statements -- they print only one value to a line. However, there is a simple way to avoid this inconvenience, which will be demonstrated in the next chapter. Even in its present form, the program is a considerable improvement over any program that does the same work without using array variables.

Using an array for a data base

We have seen how arrays are useful for storing many values of data that are to be processed. Arrays also come into use when a list or table of values must be stored for reference later on in a program. The array then serves as a data base -- a store of information that the program can draw on.

Consider the billing program we have been developing. The information for each account is read in from a card:

```
READ_LINE (ACCT_NO, OLD_BAL, PAYMENT, PURCHASE,
   INTEREST_FREE);
```

INTEREST_FREE is a BOOLEAN variable; its input value is TRUE only when no interest is to be charged to the account. ACCT_NO is an account number.

Suppose we prefer to have the computer do the work of deciding which accounts are interest-free. We take the list of interest-free accounts and punch their numbers onto cards. The billing program will then read these numbers into an array at the very beginning, and will refer to this array later to determine when INTEREST_FREE should be set to TRUE.

To implement this scheme, let us make a few assumptions:

1) There are always less than 100 interest-free accounts -- we have to set some limit, because there must be a maximum size for the array we declare;

2) The numbers are punched one to a card -- this makes it easiest to remove or add numbers when necessary;

3) The first input card contains an INTEGER value which is the number of account numbers to be read.

Only a few additions to our standard billing program are necessary. We need some new declarations:

```
VARIABLE FREE_ACCT_NO IS ARRAY(100) INTEGER,
   I, N ARE INTEGER;
```

and we need to add three statements before the main processing loop, to read in the interest-free account numbers:

```
READ_LINE (N);
IF N < 1 | N > 100 THEN TERMINATE;
FOR I THRU N REPEAT READ_LINE (FREE_ACCT_NO(I)); END;
```

Inside the main loop, we add a REPEAT statement that sets INTEREST_FREE:

```
FOR I THRU N UNTIL INTEREST_FREE REPEAT
   INTEREST_FREE := ACCT_NO = FREE_ACCT_NO(I); END;
```

If ACCT_NO matches one of the first N elements of FREE_ACCT_NO, then INTEREST_FREE is set to TRUE and the REPEAT terminates. If ACCT_NO does not match any of the first N elements of FREE_ACCT_NO, the REPEAT terminates by stepping past the limit, with INTEREST_FREE remaining FALSE.

The entire revised program looks like this:

```
VARIABLES ACCT_NO, OLD_BAL, NEW_BAL, PURCHASE, PAST_DUE,
   INTEREST, PAYMENT, I, N ARE INTEGER,
   PAST_COUNT IS INTEGER ::= 0,
```

```
   FREE_ACCT_NO IS ARRAY(100) INTEGER,
   INTEREST_FREE IS BOOLEAN;
READ_LINE (N);
IF N < 1 | N > 100 THEN TERMINATE;
FOR I THRU N REPEAT READ_LINE (FREE_ACCT_NO(I)); END;
UNTIL ACCT_NO <= 0
   REPEAT
      READ_LINE (ACCT_NO, OLD_BAL, PAYMENT, PURCHASE);
      FOR I THRU N UNTIL INTEREST_FREE REPEAT
         INTEREST_FREE := ACCT_NO = FREE_ACCT_NO(I); END;
      IF INTEREST_FREE | PAYMENT >= OLD_BAL THEN
         BEGIN;
            PAST_DUE := 0;
            INTEREST := 0;
         END;
      ELSE
         BEGIN;
            PAST_DUE := OLD_BAL - PAYMENT;
            INTEREST := PAST_DUE * 0.015;
            PAST_COUNT := PAST_COUNT + 1;
         END;
      NEW_BAL := OLD_BAL - PAYMENT + INTEREST + PURCHASE;
      PRINT (ACCT_NO, OLD_BAL, PAYMENT, PAST_DUE, INTEREST,
         PURCHASE, NEW_BAL);
   END;
PRINT (PAST_COUNT);
```

You might want to try writing this variation of the billing program without using an array variable. You will discover that it requires literally hundreds of extra statements.

Building a data base with an array

Sometimes you build up a data base in an array as you go along. An efficent prime-number generator is a good example. Our prime program in the previous chapter was somewhat wasteful: it checked for primeness by dividing each candidate number by every odd number less than the candidate's square root. It would have been enough to divide by every prime number less than the candidate's square root. But to do that we need a list of prime numbers.

The solution is to have the program make a list of prime numbers as it proceeds. The list is stored in an array. Every time a new prime is found, it is added to the array, so the list of primes is always current.

There are many ways to write a program that keeps a list of primes. The following will compute and print out up to the first 500 odd primes. The number of primes to be actually printed is read into NUMBER_OF_PRIMES at the start of the program.

```
VARIABLES CANDIDATE, TEMP, NUMBER_OF_PRIMES, I ARE INTEGER,
   N IS INTEGER ::= 0,
   PRIME IS ARRAY(500) INTEGER;
READ_LINE (NUMBER_OF_PRIMES);
IF NUMBER_OF_PRIMES < 1 | NUMBER_OF_PRIMES > 500 THEN TERMINATE;
FOR CANDIDATE FROM 3 BY 2 UNTIL N = NUMBER_OF_PRIMES
   REPEAT
   TEST:
      BEGIN;
         FOR I THRU N WHILE PRIME(I) <= CANDIDATE ** 0.5
            REPEAT
               TEMP := CANDIDATE / PRIME(I);
               IF PRIME(I) * TEMP = CANDIDATE THEN EXIT
                  FROM TEST;
            END;
         N := N + 1;
         PRIME(N) := CANDIDATE;
      END TEST;
   END;
FOR I THRU N REPEAT PRINT (PRIME(I)); END;
```

The basic plan of this program is not much different than that of our previous one. However, now the primes are stored in array PRIME, which is printed out at the end. PRIME(I) is now the divisor which is different each time through the inner loop. N keeps track of how many primes have been found so far. When a new prime is found, N is stepped by 1, and the new prime is stored in PRIME(N).

11.0

OPERATIONS ON WHOLE ARRAYS AND SUBARRAYS

In the last chapter, most of our processing of arrays was done element-by-element, using REPEAT loops and subscripted variables. We used the FOR variable as a subscript, stepping it by 1 each time through the loop; thus we were able to perform identical actions on a sequence of elements of one or more arrays.

This sort of processing of successive array elements is very common in programming. So for convenience CS-4 provides several features that make it possible to work with more than one element of an array at once, without writing REPEAT loops. You have already seen two of these features -- input and output of whole arrays. In this chapter we introduce some more: assignment of whole arrays, arithmetic operations on whole arrays, subarray subscript notation, and expressions for the sum and product over an array.

Assignment of whole arrays

For purposes of discussion in the next few sections, let us assume the following declarations:

```
VARIABLES A, B ARE ARRAY(10) INTEGER,
   C IS ARRAY(10) REAL,
   D IS ARRAY(12) INTEGER,
   I, J ARE INTEGER;
```

In the last chapter we explained how you can assign input values to all the elements of an array by writing just the array name -- with no subscript -- in a READ_LINE statement:

```
READ_LINE (A);
```

In a like manner, you can assign all the elements of one array to all the corresponding elements of a second array using the assignment operator:

```
B := A;
```

This statement has exactly the same effect as the ten assignments:

```
B(1) := A(1);
B(2) := A(2);
B(3) := A(3);
   . . .
B(9) := A(9);
B(10) := A(10);
```

or the REPEAT loop:

```
FOR I THRU 10 REPEAT B(I) := A(I); END;
```

If the elements of the two arrays are of different modes, the appropriate conversions are performed. For instance, the assignment

```
A := C;
```

causes each REAL element of C to be rounded before being assigned to the corresponding INTEGER element of A.

It is only legal to assign an array to another array of the same size. Assignments to a larger or smaller array, such as

```
A := D;
D := B;
```

will be rejected by the compiler.

Assignment of an array to a non-array variable, like

```
J := A;
```

is also prohibited. However, you are permitted to assign a non-array variable to an array, which causes the variable's single value to be assigned to every element of an array:

```
A := J;
D := B(5);
C := 1.5;
```

Here every element of C, for instance, has been assigned the value 1.5. The effect is the same as the loop

```
FOR I THRU 10 REPEAT C(I) := 1.5; END;
```

A single value may also be assigned to an array with the initialization operator. Thus one may write

```
VARIABLE C IS ARRAY(10) REAL ::= 1.5;
```

to initialize all the elements of C to 1.5.

Arithmetic operations on whole arrays

Any of the arithmetic infix operators (+ - * / **) can take two array operands of the same size. The operation is performed between all corresponding pairs of array elements. For instance, the expression

```
A + B
```

will cause ten additions to be performed:

```
A(1) + B(1)
A(2) + B(2)
   . . .
A(10) + B(10)
```

The result is an array of ten elements -- the same size as A and B -- which can be assigned to another 10-element array, like C:

```
C := A + B;
```

All conversion rules are the same as for non-array variables. If you write, for example,

```
A / B
```

the result will be an array of 10 REALs.

The prefix negation operator can also be applied to an array. It has the effect of negating each element of the array.

It is also permissible to write an infix arithmetic operator with one single INTEGER or REAL operand and one array operand. The operation will be performed between the single-value operand and each element of the array. For example, the expression

```
A * 2
```

will result in ten multiplications by 2:

```
A(1) * 2
A(2) * 2
   . . .
A(10) * 2
```

The result is an array of 10 elements, which could be assigned, say, to B:

```
B := A * 2;
```

Again, the conversion rules are analogous to those for single-value operations.

Whole-array operations can be used to write expressions that might seem a bit confusing at first. For instance, we could write:

```
A := A + A(1);
```

The compiler interprets this as an addition followed by an assignment. First it computes the array A + A(1), by performing ten additions:

```
A(1) + A(1)
A(2) + A(1)
A(3) + A(1)
   . . .
A(10) + A(1)
```

Then it assigns the ten results to the respective elements of A. The assignment causes A(1) to receive a new value; but it is the old value of A(1) that is used to compute all 10 additions.

The array-handling properties of arithmetic operators are known collectively as distributivity. Thus we say that the arithmetic operators are distributive over REAL and INTEGER arrays. You will learn in succeeding chapters that most other CS-4 operators also have distributive properties.

Subarray subscript notation

Single elements of an array are specified by subscripting the array name with a single value. In an analogous manner, a block of elements of an array -- in other words a subarray -- can be referenced by subscripting the array name with a pair of values. The two values are separated by the word AT, like this:

```
A(4 AT 2)
```

The first value indicates the number of elements in the subarray, while the second indicates the element which begins it. In this example, we have specified a subarray of 4 elements, beginning with A(2) and ending with A(5).

A subscript like (4 AT 2) is called a subarray subscript. The values in it do not have to be literals; they can be any arithmetic expressions. In general,

```
A(I AT J)
```

is a subarray of I elements, whose first element is A(J) and whose last element is A(J + I - 1). I must not be negative, and both J and J + I - 1 must be within the "bounds" dictated by the declared size of A. (In this case, J and J + I − 1 must both be between 1 and 10.)

A subarray is itself an array. In other words, a subarray of, say, 4 elements can be used just like any array of 4 elements. You can even subscript it -- for instance, the expression

```
A(4 AT 2)(3)
```

means the third element of A(4 AT 2), which is A(4).

Input and output of subarrays

In the previous chapter we encountered a problem with reading a variable number of elements into an array. In the center-of-gravity program, we read an INTEGER input value N, and then wanted to read inputs into the first N elements of array WEIGHT, and the first N elements of array DISTANCE. To do this we had to write

```
FOR I THRU N REPEAT READ_LINE (WEIGHT(I)); END;
FOR I THRU N REPEAT READ_LINE (DISTANCE(I)); END;
```

which has the disadvantage that each input value must be on a separate card (because each READ_LINE is executed N times, and each execution causes a new card to be read.)

Subarray subscripting provides a natural solution to this problem. The first N elements of WEIGHT are simply the subarray

```
WEIGHT(N AT 1)
```

which, as we have said, can be used just like any array of N elements. In particular, it can appear in a READ_LINE statement:

```
READ_LINE (WEIGHT(N AT 1));
```

which has exactly the same effect on WEIGHT as the first REPEAT above. That is, it causes N values to be read, consecutively, into WEIGHT(1) through WEIGHT(N). However, it does not require the values to be punched one to a card -- because it is a single READ_LINE statement that reads N values, rather than a loop of N statements that each read one value.

Of course, we can also read in DISTANCE(N AT 1) and eliminate the other REPEAT statement. And we can use the same subarray expressions when we print out the first N elements of WEIGHT and DISTANCE, so that the output values no longer appear one to a line.

```
VARIABLES WEIGHT, DISTANCE ARE ARRAY(25) REAL,
   WEIGHT_SUM, PRODUCT_SUM, CENTER ARE REAL,
   I, N ARE INTEGER;
REPEAT
   WEIGHT_SUM := 0;
   PRODUCT_SUM := 0;
   READ_LINE (N);
   IF N < 1 | N > 25 THEN TERMINATE;
   READ_LINE (WEIGHT(N AT 1), DISTANCE(N AT 1);
   FOR I THRU N
```

```
      REPEAT
         WEIGHT_SUM := WEIGHT_SUM + WEIGHT(I);
         PRODUCT_SUM :=
            PRODUCT_SUM + WEIGHT(I) * DISTANCE(I);
      END;
   IF WEIGHT_SUM = 0 THEN TERMINATE;
   CENTER := PRODUCT_SUM / WEIGHT_SUM
   PRINT (WEIGHT(N AT 1), DISTANCE(N AT 1), CENTER);
END;
```

Now, the program not only handles input and output better, but is shorter and simpler as well.

Using subarrays in arithmetic expressions

Subarrays can be used with the assignment and arithmetic operators. They are treated just like whole arrays: the operators are applied to all corresponding elements of the operands. Given the declarations

```
VARIABLES A, B, C ARE ARRAY(10) INTEGER,
   D IS ARRAY(4) INTEGER;
```

you are free to write statements like the following, to operate on subarrays or assign values to them:

Expression	Actions Resulting
`D := A(4 AT 4);`	assigns elements 4 through 7 of A to elements 1 through 4 of D, respectively
`C(4 AT 2) :=` `  A(4 AT 2) + B(4 AT 3);`	computes A(2) + B(3), A(3) + B(4), A(4) + B(5), and A(5) + B(6), and assigns them to C(2), C(3), C(4), and C(5)
`C(6 AT 5) :=` `  A(6 AT 1) / 2;`	divides elements 1 through 6 of A by 2, and assigns the results to elements 5 through 10 of C

Note that the array size rules are strictly observed. It is not legal, for instance, to write

```
D := A(3 AT 1);
```

because D has 4 elements, while A(3 AT 1) has 3 elements.

An example with subarray calculations

Subarray calculations and assignments can sometimes be used to advantage in writing programs to perform numerical algorithms. As an elementary example, let us construct a program to print a table of binomial coefficients -- also known as Pascal's Triangle. The first seven rows of the triangle look like this:

row number \ column number	1	2	3	4	5	6	7	...
1	1							
2	1	1						
3	1	2	1					
4	1	3	3	1				
5	1	4	6	4	1			
6	1	5	10	10	5	1		
7	1	6	15	20	15	6	1	
⋮								

Row number N always has N entries. Its easy to calculate the elements in any row, once you know the row above it. The element in the first column is always 1. Every other element is the sum of the number directly above it, and the number diagonally above it to the left. (The 1 at the far right of each row doesn't have a number above it; but if you imagine there is a zero above, the rule still holds.)

Before writing a program, it helps to state the rules more formally. Assume you already have row N-1, for $N \geq 2$; then the following steps produce row N:

Step 1: add a zero in column N of row N-1.
Step 2: row N, column 1 is assigned 1.
Step 3: row N, column I is assigned row N-1, column I + row N-1, column I-1, for $2 \leq I \leq N$.

Now, it is not difficult to write a CS-4 statement to perform these steps. We declare an INTEGER array, named PASCAL. Assume this array "contains" row N-1 -- that is, its first N-1 elements are the entries of row N-1, and the rest of its elements are zeros. We want to change its elements so that it "contains" row N instead.

Step 1 already is satisfied, since PASCAL(N) = 0. Step 2 says that PASCAL(1), which is already 1, remains unchanged. Step 3 says we make the following assignments to PASCAL(2) through PASCAL(N):

```
PASCAL(I) := PASCAL(I) + PASCAL(I - 1);   2≤I≤N
```

With subarray subscript notation, this can be written as a single array assignment:

```
PASCAL(N - 1 AT 2) := PASCAL(N-1 AT 2) + PASCAL(N-1 AT 1);
```

This single statement transforms the contents of PASCAL from row N - 1 to row N. Then we can print out row N by printing out the first N elements:

```
PRINT (PASCAL(N AT 1));
```

After printing out row N we need only step N by 1 and repeat the transformation to compute the next row. A complete program to print out rows 1 through 20 looks like this:

```
VARIABLE N IS INTEGER,
   PASCAL IS ARRAY(20) INTEGER ::= 0;
PASCAL(1) := 1;
PRINT (PASCAL(1));
FOR N FROM 2 THRU 20 REPEAT
   REPEAT
      PASCAL(N - 1 AT 2) := PASCAL(N - 1 AT 2) +
         PASCAL(N - 1 AT 1);
      PRINT (PASCAL(N AT 1));
   END;
```

The first two assignments are necessary to initialize PASCAL to row 1.

Subarray addition and assignment is especially powerful in this case. You should be able to satisfy yourself that there is no way to write a REPEAT loop of the form

```
FOR I FROM 2 THRU N REPEAT ... END;
```

which does the same work without subarray operations, unless you declare some additional temporary storage.

Summing over an array

There still is one element-by-element REPEAT loop in our center-of-gravity program. It is the one that does the summing:

```
FOR I THRU N
   REPEAT
      WEIGHT_SUM := WEIGHT_SUM + WEIGHT(I);
      PRODUCT_SUM :=
         PRODUCT_SUM + WEIGHT(I) * DISTANCE(I);
   END;
```

This loop, too, can be eliminated by use of a convenient and brief CS-4 expression. It consists of the name SUM followed by an array name in parentheses:

```
SUM (WEIGHT)
```

Its value is equal to the sum of all the array-element values (the sum "over the array", so to speak). Thus it represents an INTEGER value for an INTEGER array, and a REAL value for a REAL array. You can use it just like any INTEGER or REAL variable, in expressions like

```
WEIGHT_SUM := SUM (WEIGHT);
IF SUM (WEIGHT) = 0 THEN TERMINATE;
```

The first statement assigns the sum over WEIGHT to WEIGHT_SUM. The second executes TERMINATE if the sum over WEIGHT is zero.

SUM (WEIGHT) is the sum of all elements of WEIGHT. In our program, we want the sum of just the first N elements of WEIGHT -- the sum over WEIGHT(N AT 1):

```
SUM (WEIGHT(N AT 1))
```

We also want to find the sum of the first N weight-distance products. We can do so by distributing * over WEIGHT(N AT 1) and DISTANCE(N AT 1) to compute an array of weight-distance products:

```
WEIGHT(N AT 1) * DISTANCE(N AT 1)
```

The sum of the products is the sum over this array:

```
SUM (WEIGHT(N AT 1) * DISTANCE(N AT 1))
```

With these last two SUM expressions we can shorten and simplify the center-of-gravity program still further:

```
VARIABLES WEIGHT, DISTANCE ARE ARRAY(25) REAL,
   WEIGHT_SUM, PRODUCT_SUM, CENTER ARE REAL,
   N IS INTEGER;
REPEAT
   READ_LINE (N);
   IF N < 1 | N > 25 THEN TERMINATE;
   READ_LINE (WEIGHT(N AT 1), DISTANCE(N AT 1));
   WEIGHT_SUM := SUM (WEIGHT(N AT 1));
   IF WEIGHT_SUM = 0 THEN TERMINATE;
   PRODUCT_SUM :=
      SUM (WEIGHT(N AT 1) * DISTANCE(N AT 1));
   CENTER := PRODUCT_SUM / WEIGHT_SUM;
   PRINT (WEIGHT(N AT 1), DISTANCE(N AT 1), CENTER);
END;
```

Not only is the inner REPEAT loop eliminated; we are also saved the trouble of initializing WEIGHT_SUM and PRODUCT_SUM to zero before each input case.

The product over an array

There is also an expression, analogous to SUM, for the value obtained by multiplying together all the elements of an array. It uses the name PRODUCT:

```
PRODUCT (VALUES)
```

As an example, if VALUES here were an ARRAY(5) INTEGER, then PRODUCT (VALUES) would be an INTEGER equal to

```
VALUES(1) * VALUES(2) * VALUES(3) * VALUES(4) * VALUES(5)
```

12.0

BOOLEAN ARRAYS

You have now seen most of the tools for working with arrays in CS-4. They include:

the ARRAY mode generator
single and subarray subscripts
array input, output, and assignment
distributivity of operators
expressions like SUM and PRODUCT

We introduced these tools in working with REAL and INTEGER arrays. In this chapter, we show how to use them with BOOLEAN arrays as well.

General properties of arrays applied to BOOLEAN mode

You already know quite a few things about BOOLEAN arrays -- just from your knowledge of the properties of arrays in general. Here is a refresher:

1) Introducing the ARRAY mode generator in Chapter 10, we said it could be used with any element-mode. Hence we can use ARRAY to declare arrays of BOOLEANs:

```
VARIABLE CHECK IS ARRAY(100) BOOLEAN;
VARIABLES TEST1, TEST2 ARE ARRAY(5) BOOLEAN;
```

2) The rules for subscripting hold for any array declared with ARRAY, whatever the element-mode. So you know that

```
CHECK(43)
TEST2(1)
```

are subscripted variable names with single BOOLEAN values; and

```
CHECK(5 AT 11)
```

is a subarray of the same size as TEST1.

3) The properties of READ_LINE and PRINT hold for arrays of any mode. So you can write statements like

```
READ_LINE (TEST2);
PRINT (CHECK(10 AT 91));
```

to read five BOOLEAN values into TEST2, and print the last ten values from CHECK.

4) Assignment for any mode can be extended to arrays of that mode declared with ARRAY. You know already that a single BOOLEAN variable can be assigned a single BOOLEAN value. So it follows that a BOOLEAN array variable can also be assigned a single BOOLEAN value (in which case the value is assigned to each element of the array), or it can be assigned a BOOLEAN array value of matching size.

These tools -- ARRAY, subscripting, array input/output, array assignment -- are all part of the general concept of "array" in CS-4. You can use them with INTEGER, REAL, or BOOLEAN -- or any other mode.

A BOOLEAN array as a look-up table

We can use a BOOLEAN array to improve upon array use in our last version of the billing program (toward the end of Chapter 10). That example defined an INTEGER array called FREE_ACCT_NO, into which we read all the numbers of interest-free accounts. Then, in the main loop, we checked the number of each input account by comparing it with the numbers in the FREE_ACCT_NO array. This is the relevant part of the program:

```
VARIABLES ACCT_NO, OLD_BAL, NEW_BAL, PURCHASE, PAST_DUE,
   INTEREST, PAYMENT, I, N ARE INTEGER,
   PAST_COUNT IS INTEGER ::= 0,
   INTEREST_FREE IS BOOLEAN,
   FREE_ACCT_NO IS ARRAY(100) INTEGER;
READ_LINE (N);
IF N < 1 | N > 100 THEN TERMINATE;
FOR I  THRU N REPEAT READ_LINE (FREE_ACCT_NO(I)); END;
UNTIL ACCT_NO <= 0
```

```
REPEAT
   READ_LINE (ACCT_NO, OLD_BAL, PAYMENT, PURCHASE);
   FOR I THRU N UNTIL INTEREST_FREE REPEAT
      INTEREST_FREE := ACCT_NO = FREE_ACCT_NO(I); END;
   IF INTEREST_FREE | PAYMENT >= OLD_BAL THEN
   . . .
```

N is the number of interest-free accounts; it is read from the first input card.

The drawback in this approach lies in the inner REPEAT loop:

```
FOR I THRU N UNTIL INTEREST_FREE REPEAT
   INTEREST_FREE := ACCT_NO = FREE_ACCT_NO(I); END;
```

It loops N times for every non-interest-free account, and between 1 and N times for every interest-free account. Unless N is very small, that adds up to quite a lot of time spent looping.

What we need is a scheme that avoids having to "search" through the whole list of account numbers every time we process an account. We could create a sort of table, in which we can "look up" an account number to see if its account is interest-free, without having to do a lot of comparisons.

In practice, how do we create such a table? By declaring BOOLEAN array. Let's say account numbers range from 1 to 999; then we define FREE_ACCT_TABLE as a BOOLEAN array of 999 elements:

```
VARIABLE FREE_ACCT_TABLE IS ARRAY(999) BOOLEAN;
```

We initialize all elements of FREE_ACCT_TABLE to FALSE; then we set the elements corresponding to interest-free accounts to TRUE. If account number 50, for instance, were interest-free, FREE_ACCT_TABLE(50) would be TRUE; otherwise FREE_ACCT_NO(50) would be FALSE.

Now, in the main REPEAT loop, FREE_ACCT_TABLE serves as a table. Every time a value of ACCT_NO is read, we check FREE_ACCT_TABLE(ACCT_NO). If it's FALSE, we compute interest (if any) for the account; if it's TRUE, we don't. Each element of the array serves as an entry in our conceptual table. We can look up individual elements -- or entries -- without searching through other entries. The beginning of the rewritten billing program looks like this:

```
VARIABLES FREE_ACCT, ACCT_NO, OLD_BAL, NEW_BAL, PURCHASE,
   PAST_DUE, INTEREST, PAYMENT, I, N ARE INTEGER,
   PAST_COUNT IS INTEGER ::= 0,
   FREE_ACCT_TABLE IS ARRAY(999) BOOLEAN ::= FALSE;
READ_LINE (N);
FOR I THRU N
   REPEAT
      READ_LINE (FREE_ACCT);
      IF FREE_ACCT < 1 | FREE_ACCT > 999 THEN TERMINATE;
      FREE_ACCT_TABLE(FREE_ACCT) := TRUE;
   END;
UNTIL ACCT_NO <= 0
   REPEAT
      READ_LINE (ACCT_NO, OLD_BAL, PAYMENT, PURCHASE);
      IF ACCT_NO < 1 | ACCT_NO > 999 THEN TERMINATE;
      IF FREE_ACCT_TABLE(ACCT_NO) | PAYMENT >= OLD_BAL THEN
      . . .
```

Note the use of whole-array assignment to initialize FREE_ACCT_TABLE; and the use of a single element of that array in a boolean expression in the last IF statement.

To be fair, we ought to mention that the revised program has a drawback, too: it requires an array of 999 elements, which might well take up more space than the 100-element array of the old program. If account numbers had six digits, as they often do, an array of a million BOOLEAN elements would be required to guarantee a place in the table for every possible account number -- even if only a fraction of the possible account numbers were in use at any one time. In such a case, economy in execution speed is achieved only by using more storage space -- which may also be costly. Tradeoffs of this sort, between speed and size, are common considerations in programming. Often the best implementations employ some form of compromise, as we shall see in more advanced examples.

Distributivity of comparison operators over REAL and INTEGER values

All the CS-4 comparison operators are distributive -- they can take one or two REAL or INTEGER array operands and yield a BOOLEAN array result. In more detail, the rules are:

1) If both operands are REAL or INTEGER arrays, they must be of the same size. Corresponding elements of the two arrays are compared, and the result is a BOOLEAN array of the same size as the operands.

2) If one operand is a single INTEGER or REAL, the other an INTEGER or REAL array, the single operand is compared with each element of the array. The result is a BOOLEAN array of the same size as the array operand.

3) INTEGER-to-REAL conversion is performed before each comparison of an INTEGER value with a REAL value.

Here are a few examples. We can compare two INTEGER arrays and assign the result to a BOOLEAN array:

```
VARIABLES TEST, DATA ARE ARRAY(10) INTEGER,
   COMPARE IS ARRAY(10) BOOLEAN;
COMPARE := TEST ~= DATA;
```

The comparison of TEST and DATA causes ten comparison operations to be performed on their elements:

```
TEST(1)  ~= DATA(1)
TEST(2)  ~= DATA(2)
. . .
TEST(10) ~= DATA(10)
```

The result is an array of ten BOOLEAN elements, which are assigned to the corresponding elements of COMPARE. The expression

```
DATA > 3.14
```

also results in 10 comparisons yielding an ARRAY(10) BOOLEAN:

```
DATA(1)  > 3.14
DATA(2)  > 3.14
. . .
DATA(10) > 3.14
```

Each comparison is between an INTEGER and a REAL, so the INTEGER values in DATA are converted to REALs before the comparison.

Distributivity of comparison operators over BOOLEAN values

The operators = and ~= can be applied to BOOLEAN operands, and they are distributive over BOOLEAN arrays. Thus comparison of two equal-sized BOOLEAN arrays, or a BOOLEAN array and a single BOOLEAN yields a BOOLEAN array of the same size as the array operand(s). The rules for actually applying the operators are the usual ones.

Distributivity of logical operators

As you have probably guessed by now, the CS-4 logical operators are also distributive. The infix operators | and & accept one or two BOOLEAN arrays as operands, and produce a BOOLEAN array; and the prefix operator ~ can operate on and produce a BOOLEAN array. The exact rules are analogous to those for the other distributive operators, so we won't belabor them here.

ALL and ANY expressions

In the last chapter we showed how it was convenient to have an expression for the result of adding or multiplying together all the elements of a REAL or INTEGER array. There are similar, equally useful expressions for the result of ANDing or ORing together the elements of a BOOLEAN array.

The expression for applying the operator & between all elements of an array uses the name ALL. For instance,

```
ALL (COMPARE)
```

has the value of

```
COMPARE(1) & COMPARE(2) & ... & COMPARE(10)
```

which is TRUE only if all elements of COMPARE are TRUE (hence the name ALL). ALL is applied to an array, but its result is a single BOOLEAN value (just as SUM is applied to an array and yields a single REAL or INTEGER value).

The name ANY is used analogously, for the result of applying the operator | between all elements of an array. The expression

```
ANY (COMPARE)
```

has the same value as

```
COMPARE(1) | COMPARE(2) | ... | COMPARE(10)
```

which is TRUE if any one element of COMPARE is TRUE (hence the name ANY).

An example

The concepts introduced in the last three sections are often most valuable when they are used together. As an example, we consider the problem of adding some checking to the center-of-gravity problem.

Suppose the rod on which the weights are hung is ten feet long, so that a weight cannot be more than five feet either way from the center of the rod. Then we want to write a test to check that no DISTANCE value is greater than 5 or less than -5. Remembering that N is the number of weights whose distances are being measured, we can write

```
DISTANCE(N AT 1) > 5
```

to compare the N DISTANCE inputs with 5, and similarly

```
DISTANCE(N AT 1) < -5
```

to compare them with -5. Each of these expressions yields a BOOLEAN array of N elements, so we can OR the results together:

```
DISTANCE(N AT 1) > 5 | DISTANCE(N AT 1) < -5
```

This expression also yields a BOOLEAN array of N elements; the Ith element of this resultant array is TRUE only if DISTANCE(I) is out of bounds (greater than 5 or less than -5). We have an error if any element of DISTANCE(N AT 1) is out of bounds, that is, if

```
ANY (DISTANCE(N AT 1) > 5 | DISTANCE(N AT 1) < -5)
```

So we can test for an error with a single IF statement:

```
IF ANY (DISTANCE(N AT 1) > 5 | DISTANCE(N AT 1) < -5)
   THEN TERMINATE;
```

While we're at it, we might as well also test

```
ANY (WEIGHT(N AT 1) < 0)
```

to avoid negative weights, and

```
ALL (WEIGHT(N AT 1) = 0)
```

to make sure that WEIGHT_SUM won't be zero. Putting all these expressions together, the program looks like this:

```
VARIABLES WEIGHT, DISTANCE ARE ARRAY(25) REAL,
   WEIGHT_SUM, PRODUCT_SUM, CENTER ARE REAL,
   N IS INTEGER;
REPEAT
   READ_LINE (N);
   IF N < 1 | N > 25 THEN TERMINATE;
   READ_LINE (WEIGHT(N AT 1), DISTANCE(N AT 1));
   IF ANY (DISTANCE(N AT 1) > 5 |
           DISTANCE(N AT 1) < -5) |
      ANY (WEIGHT(N AT 1) < 0) |
      ALL (WEIGHT(N AT 1) = 0)
      THEN TERMINATE;
   WEIGHT_SUM := SUM (WEIGHT(N AT 1));
   PRODUCT_SUM :=
      SUM (WEIGHT(N AT 1) * DISTANCE(N AT 1));
   CENTER := PRODUCT_SUM / WEIGHT_SUM;
   PRINT (WEIGHT(N AT 1), DISTANCE(N AT 1), CENTER);
END;
```

13.0

MULTI-DIMENSIONAL ARRAYS

This chapter widens still further the family of arrays to which the standard tools can be applied. Just as we naturally extended the concept of array to BOOLEAN arrays in the last chapter, here we demonstrate the use of so-called "multi-dimensional" arrays in CS-4.

Multi-dimensional arrays can be thought of simply as arrays that are numbered in more than one direction, or along more than one axis. They are often used to program problems involving surfaces or spaces; it's the geometrical analogy that gives them their name.

Multi-dimensional arrays can also be thought of as arrays whose elements are arrays. They can be seen as a natural outgrowth of CS-4's general definition of the ARRAY mode generator.

This chapter introduces multi-dimensional arrays first through the geometrical approach, because it is somewhat more intuitive. The array of arrays approach is also outlined, after the basic rules have been established.

Declaring multi-dimensional arrays

The principle behind multi-dimensional arrays is fairly straightforward, so we will introduce them briefly before showing what can be done with them.

Each element of a multi-dimensional array is specified by a list of two or more subscripts. The subscripts are separated by commas. For example, if TABLE were a 2-dimensional array, then each element of TABLE would be written with two subscripts, in this manner:

```
TABLE(2,7)
TABLE(5,5)
TABLE(1,10)
TABLE(7,2)
```

These are all different elements of TABLE -- in particular, TABLE(7,2) is different from TABLE(2,7).

As before, a declaration statement must state the number and mode of the elements of TABLE. In this case, the declaration might have been

```
VARIABLE TABLE IS ARRAY(8,12) INTEGER;
```

This says that the first subscript of TABLE ranges from 1 to 8, and the second subscript ranges from 1 to 12. There will be an element of TABLE for every combination of first subscript and second subscript; in all, 8 * 12 = 96 elements, of INTEGER mode.

The same reasoning extends to arrays of higher dimensions. For instance, this statement declares a 4-dimensional array:

```
VARIABLE HYPER IS ARRAY(10,2,20,20) BOOLEAN;
```

Each element of HYPER is specified by a different list of four subscripts:

```
HYPER(2,2,1,5)
HYPER(1,2,20,19)
HYPER(10,1,1,1)
```

The first subscript may range from 1 to 10, the second from 1 to 2, the third and fourth from 1 to 20. In all there are 10 * 2 * 20 * 20 = 8000 elements. Each element is a BOOLEAN.

A two-dimensional problem

We will start off with a problem that uses two-dimensional arrays to store and calculate measurements at points on a two-dimensional surface. Our surface, diagrammed in Figure 13-1, is a sheet of metal or some other material that conducts heat. It is 5 inches wide and 8 inches long. We assume that, initially, the entire sheet has a temperature of 0 degrees. Now, suppose we can change the heat distribution along the edges, so that some parts of the edges are kept warmer than other parts; the problem is to write a program that predicts what the temperatures will be on the interior of the sheet, after the heat has spread itself out, or "reached an equilibrium."

Of course, it's impossible to measure the temperature at every point on the sheet, because there's an infinite number of points. What we do is mark out a grid of horizontal and vertical lines on the sheet (see Figure 13-1) and restrict the problem to measuring the temperatures where the grid lines intersect. There are nine rows of six grid points

each; common mathematical notation represents the temperature of a grid point by using <u>two</u> subscripts, one for the row number and one for the column number, so that

$$t_{ij}, \quad 1 \leq i \leq 9, \quad 1 \leq j \leq 6$$

stands for the temperature of the grid point in the jth column and the ith row. The temperature of the point circled in Figure 13-1, for example, would be t_{35}. Initially, we are assuming all $t_{ij} = 0$.

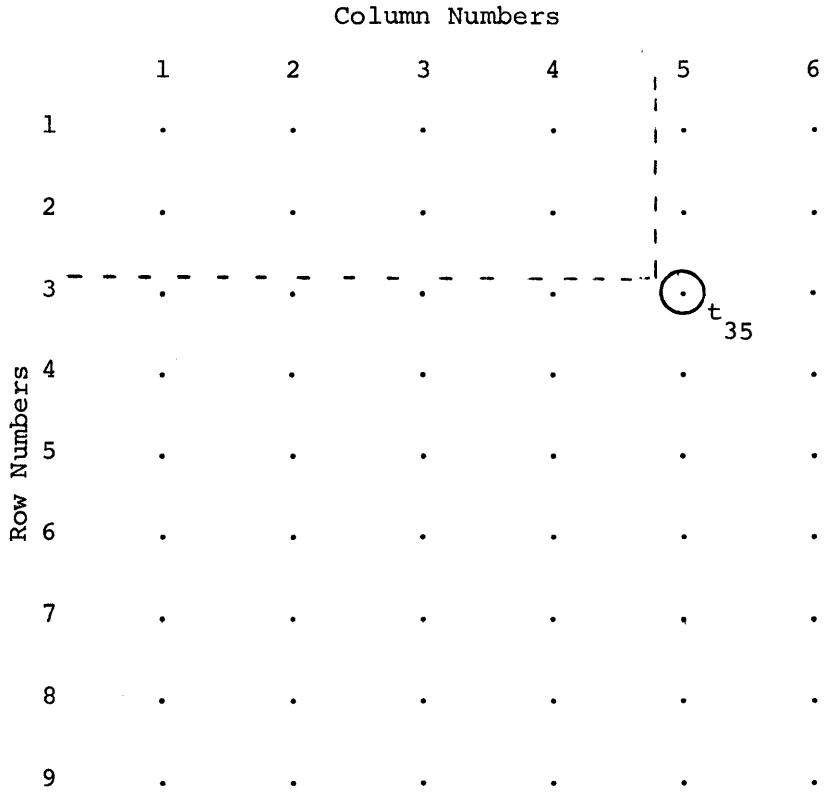


Figure 13-1

The temperature grid. The temperature of the circled point -- the 3rd row and 5th column -- is t_{35}.

Next heat is applied, at the edge points. These points are indicated with x's in Figure 13-2. (The corners are special cases -- we'll ignore them for the present.) Using the t_{ij} notation, applying heat at the edges means changing the values of the following t's:

t_{1j}, $2 \le j \le 5$
t_{9j}, $2 \le j \le 5$
t_{i1}, $2 \le i \le 8$
t_{i6}, $2 \le i \le 8$

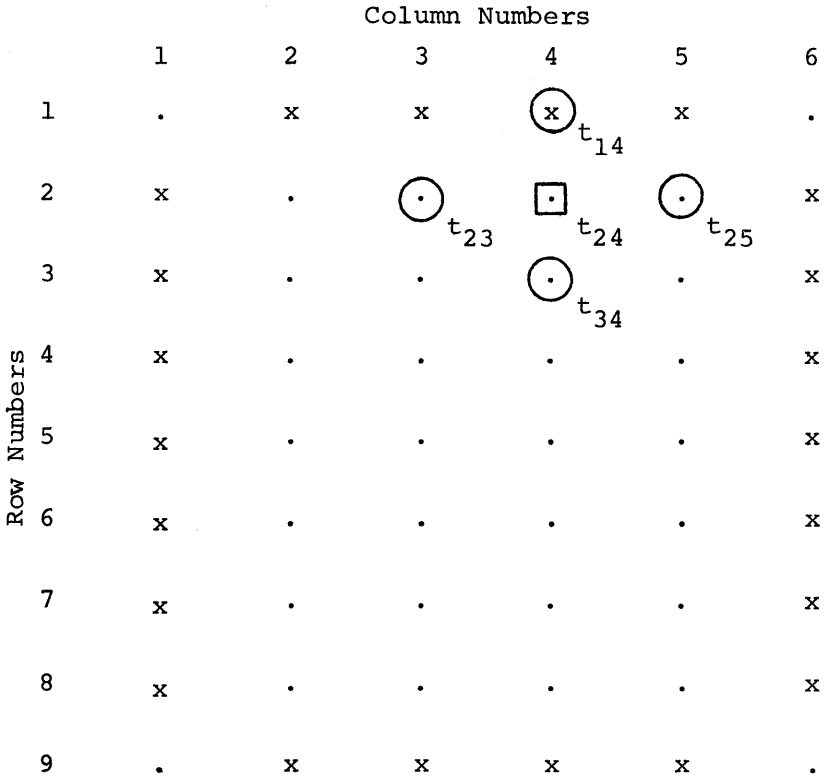


Figure 13-2

Boundary and neighbor points. The boundary points are marked with x's. The point with temperature t_{24} -- marked with a square -- is shown surrounded by its four neighbor points, which are circled.

As the heat spreads, the values of the interior t's also change. To find the new interior t's, we apply the following simple model: the temperature at a grid point must be the average of the temperatures of the four points around it. For instance, look at t_{24} in Figure 13-2. By our model, its temperature at equilibrium must be

$$t_{24} = (t_{14} + t_{34} + t_{23} + t_{25}) / 4$$

Initially, all t's are zero, so all points are in equilibrium. But after the edges are heated, the interior points are out of equilibrium, so their temperatures all have to be recalculated as follows:

$$t_{ij} = (t_{(i-1)j} + t_{(i+1)j} + t_{i(j-1)} + t_{i(j+1)}) / 4,$$
$$2 \le i \le 8,\ 2 \le j \le 5$$

Is this all that has to be done? Probably not -- probably, after just one recalculation, the t's are still out of equilibrium. To see why, consider t_{24} again. The <u>new</u> value of t_{24} is the average of the <u>old</u> values of its neighbors, t_{14}, t_{34}, t_{23}, and t_{25}. But t_{34}, t_{23}, and t_{25} have also acquired new, recalculated values (t_{14}, remember, is a fixed boundary point). So the new t_{24} may still be out of equilibrium, with the new values of its neighbors; and we will have to recalculate t_{24}, and all the other t's, yet again. The recalculation of the t's must go on indefinitely, until every t_{ij} is in equilibrium with its neighbors.

You can imagine the whole process intuitively as follows. If any t_{ij} is cooler than the average of its neighbors, heat will flow to it to make up the deficiency. So it will get warmer. While a point is warming up, its neighbors may also be warming up; in that case the point's temperature has to increase still further. Heat continues flowing until every interior point of the sheet has reached equilibrium; then the sheet's temperature is again stable (as it was when all t's were 0).

Of course, in reality the heating process proceeds continuously over time, not in discrete steps. But our model approximates the real process well enough so that the end results are virtually the same.

(What about the corner points? They play no part in our calculations -- because they are not neighbors to any interior points. The best we can come to guessing their temperatures is by assigning them the average of the two closest edge points. So we'll have our program set

$$t_{11} = (t_{12} + t_{21}) / 2$$

and similarly for the other three corners.)

Programming the problem with 2-dimensional arrays

By now it should be obvious that the double-subscript notation we've been using for grid points on a plane should translate very nicely into double-subscript, 2-dimensional arrays in CS-4. It is possible to program the problem with 1-dimensional arrays -- but only at a cost in simplicity and clarity.

Accordingly, our sample program begins by declaring two two-dimensional array variables. The first subscript of each varies from 1 to 9, the second from 1 to 6:

```
VARIABLES TEMPERATURE, NEW_TEMPERATURE ARE ARRAY(9,6) REAL;
```

TEMPERATURE(I,J) represents the temperature of the metal sheet at the grid point in the jth column of the ith row -- what we called t_{ij} in the previous section. NEW_TEMPERATURE holds the corresponding "new t's" that are calculated in finding the equilibrium.

The executable part of the program divides logically into three parts. First, an initialization section reads in the boundary temperatures and sets the interior temperatures to zero. Second, a main loop repeatedly calculates each interior temperature as the average of its neighbors, until equilibrium is reached. Third, a conclusion section assigns temperatures to the four corner points, and prints out the whole equilibrium array of temperatures.

Altogether, the program looks like this:

```
VARIABLES TEMPERATURE, NEW_TEMPERATURE ARE ARRAY(9,6) REAL,
   I, J ARE INTEGER,
   EQUILIBRIUM IS BOOLEAN;
FOR I FROM 2 THRU 8 REPEAT
   FOR J FROM 2 THRU 5 REPEAT
      TEMPERATURE(I,J) := 0; END; END;
FOR I FROM 2 THRU 8 REPEAT
   READ_LINE (TEMPERATURE(I,1)); END;
FOR I FROM 2 THRU 8 REPEAT
   READ_LINE (TEMPERATURE(I,6)); END;
FOR J FROM 2 THRU 5 REPEAT
   READ_LINE (TEMPERATURE(1,J)); END;
FOR J FROM 2 THRU 5 REPEAT
   READ_LINE (TEMPERATURE(9,J)); END;
UNTIL EQUILIBRIUM
   REPEAT
      EQUILIBRIUM := TRUE;
      FOR I FROM 2 THRU 8
         REPEAT
            FOR J FROM 2 THRU 5
               REPEAT
                  NEW_TEMPERATURE(I,J) :=
                     (TEMPERATURE(I-1,J) + TEMPERATURE(I+1,J) +
                      TEMPERATURE(I,J-1) + TEMPERATURE(I,J+1)) / 4;
                  IF NEW_TEMPERATURE(I,J) - TEMPERATURE(I,J)
                     > 0.001 |
                     NEW_TEMPERATURE(I,J) - TEMPERATURE(I,J)
                     < -0.001
                     THEN EQUILIBRIUM := FALSE;
               END;
         END;
      IF ~EQUILIBRIUM THEN
         FOR I FROM 2 THRU 8 REPEAT
            FOR J FROM 2 THRU 5 REPEAT
               TEMPERATURE(I,J) := NEW_TEMPERATURE(I,J); END; END;
   END;
TEMPERATURE(1,1) := (TEMPERATURE(1,2) + TEMPERATURE(2,1)) / 2;
TEMPERATURE(1,6) := (TEMPERATURE(1,5) + TEMPERATURE(2,6)) / 2;
TEMPERATURE(9,1) := (TEMPERATURE(8,1) + TEMPERATURE(9,2)) / 2;
TEMPERATURE(9,6) := (TEMPERATURE(9,5) + TEMPERATURE(8,6)) / 2;
FOR I FROM 1 THRU 9
   REPEAT
```

```
            PRINT (TEMPERATURE(I,1), TEMPERATURE(I,2),
                   TEMPERATURE(I,3), TEMPERATURE(I,4),
                   TEMPERATURE(I,5), TEMPERATURE(I,6));
        END;
```

The critical statement in the inner loop is the one that compares TEMPERATURE and NEW_TEMPERATURE:

```
IF NEW_TEMPERATURE(I,J) - TEMPERATURE(I,J) > 0.001 |
   NEW_TEMPERATURE(I,J) - TEMPERATURE(I,J) < -0.001
   THEN EQUILIBRIUM := FALSE;
```

EQUILIBRIUM is reinitialized to TRUE at the beginning of each loop. This statement says: if the old and new temperatures at any grid point differ by more than 0.001, then equilibrium has not yet been reached; so set EQUILIBRIUM to FALSE. The value 0.001 is not important in itself; it's an arbirtrary standard we've picked for this particular problem. It's tempting to try to simplify the IF statement by dispensing with an equilibrium standard, and writing

```
IF NEW_TEMPERATURE(I,J) ~= TEMPERATURE(I,J) THEN
   EQUILIBRIUM := FALSE;
```

But, as we explained in Chapter 6, comparisons of REALs are somewhat unpredictable. In this particular case, there is a definite risk that the main loop will be an infinite loop, unless we specify some "fuzz" -- such as 0.001 -- that indicates when two temperatures are so close that they can be considered equal for our purposes.

To keep the sample program simple, we restricted it to processing one 9-by-6 case per run. It can be generalized, though, to handle many cases in one run, and to take input for grids of varying length and width. And a similar program can be written to handle the analogous problem for 3-dimensional blocks, using 3-dimensional arrays and six neighbors per interior point.

Simplifying multi-dimensional array expressions

Our sample program unintentionally demonstrates something about programming with arrays of several dimensions: element-by-element processing of multi-dimensional arrays requires a lot of nested REPEAT loops and long, bulky expressions. Tools for manipulating all or part of an array at once could do a lot to shrink and simplify a program -- even more than they do for a program using 1-dimensional arrays.

Fortunately, all the array-handling tools introduced in previous chapters extend in a natural way to multi-dimensional arrays. In succeeding sections, we will show how. But before we do so, we must define some more precise terminology.

The number of dimensions a variable has is called its rank. Thus, the arrays we dealt with in Chapters 10-12 all had a rank of 1; TEMPERATURE in our latest example has a rank of 2. A variable that represents a single value, like EQUILIBRIUM or I or TEMPERATURE(I,J), is said to have a rank of zero.

We also talk about the size of each dimension of an array. The first dimension of TEMPERATURE, for instance, has a size of 9, while the second dimension has a size of 6. As we mentioned in the first section of this chapter, the total number of elements in an array is found by multiplying together the sizes of all its dimensions. Be sure you understand the difference between dimension size and total number of elements. It's possible to declare two different arrays, such as

```
VARIABLE TABLE_A IS ARRAY(6,3,2) INTEGER,
   TABLE_B IS ARRAY(2,2,9) INTEGER;
```

which have the same element total (36), but which differ in the size of each of their dimensions. (Of course, for a 1-dimensional array, the size of its single dimension is the same as the total number of its elements; that's why we could use "size" and "number of elements" interchangeably in previous chapters.)

The dimension sizes and rank of an array, taken together, are called its arrayness. Thus, two arrays have the same arrayness only when they have the same rank, and their corresponding dimensions have the same sizes. In the special case of 1-dimensional arrays, equal size always implies equal arrayness. But this is not true in general; for instance these two 35-element arrays:

```
VARIABLE BLOCK1 IS ARRAY(5,7) REAL,
   BLOCK2 IS ARRAY(7,5) REAL;
```

do not have the same arrayness, because BLOCK1 has a first-dimension size of 5, while BLOCK2 has a first-dimension size of 7. (Nor do they have the same second dimension sizes.)

Whole-array assignment and input/output

One array can be assigned to another with the assignment operator, provided they have the same arrayness. The assignment causes each element of the right-hand operand to be assigned to the corresponding element of the left-hand operand. For instance, given the declarations in the temperature-grid program, we could write

```
TEMPERATURE := NEW_TEMPERATURE;
```

which would cause the 54 assignments:

```
TEMPERATURE(I,J) := NEW_TEMPERATURE(I,J); 1 ≤ I ≤ 9, 1 ≤ J ≤ 6
```

One can also assign a single value to all the elements of an array of any rank, with either the assignment or initialization operator. For example, either this declaration:

```
VARIABLE TEMPERATURE IS ARRAY(9,6) REAL ::= 0;
```

or this assignment:

```
TEMPERATURE := 0;
```

sets all 54 elements of TEMPERATURE to zero.

Values can be read into all the elements of an array by writing its name in a READ_LINE statement:

```
READ_LINE (TEMPERATURE);
```

Since the input to READ_LINE is always a one-dimensional stream of values, there must be some rule that tells in what order the elements of a multi-dimensional array receive their assignments. The rule is this: elements with a given first subscript are assigned values before elements having a larger first subscript; among elements having the same first subscript, those with a given second subscript are read in before those with a larger second subscript; among elements with the same second subscript, the ordering depends on the third subscript; and so on. The READ_LINE above would thus assign 54 values to elements of TEMPERATURE in the following order:

```
TEMPERATURE(1,1)
TEMPERATURE(1,2)
TEMPERATURE(1,3)
...
TEMPERATURE(1,6)
TEMPERATURE(2,1)
TEMPERATURE(2,2)
...
TEMPERATURE(2,6)
TEMPERATURE(3,1)
...
TEMPERATURE(6,6)
```

Again, for a 3-dimensional array

```
VARIABLE TAB IS ARRAY(2,2,3) BOOLEAN;
READ_LINE (TAB);
```

12 BOOLEAN values are read into elements of TAB in this order:

```
TAB(1,1,1)
TAB(1,1,2)
TAB(1,1,3)
TAB(1,2,1)
TAB(1,2,2)
TAB(1,2,3)
TAB(2,1,1)
TAB(2,1,2)
TAB(2,1,3)
TAB(2,2,1)
TAB(2,2,2)
TAB(2,2,3)
```

The process of ordering the elements of an array into a one-dimensional list is called unraveling. One good way to remember the unraveling rule for multi-dimensional arrays is to think of it as a numerical equivalent of alphabetizing: first you compare elements in the first (subscript) place, then in the second, then in the third, and so forth until you find a mismatch; the order of two elements is determined by the numerical order of their subscripts in the first place where they don't match.

Another way to remember the rule is to think of the subscripts varying like the numbers on the odometer of a car: each place is held constant while the subscript to its right goes through its entire range.

Whole arrays can be printed out with the PRINT statement:

```
PRINT (TEMPERATURE);
```

The order in which the elements are printed out is also determined by the unraveling rules.

Subarray subscripting

The use of subarray subscripts extends naturally and conveniently to multi-dimensional arrays. For a two-dimensional array, such as

```
VARIABLE A IS ARRAY(8,6) INTEGER;
```

we can illustrate the rules with a row-and-column diagram like the one in Figure 13-3.

Two single subscripts:

```
A(3,6)
```

yield, as you already know, a value of rank zero -- the INTEGER value of a single element. In the diagram, this element is found at the intersection of row 3 and column 6.

Suppose the first subscript is still single, but the second is a subarray subscript; for example:

```
A(2,4 AT 2)
```

This subscripted variable names an array of rank 1 and size four, whose elements are identical with, respectively: A(2,2), A(2,3), A(2,4), and A(2,5). In the diagram, this array is represented as a series of elements -- numbers 2 through 5 -- in the second row.

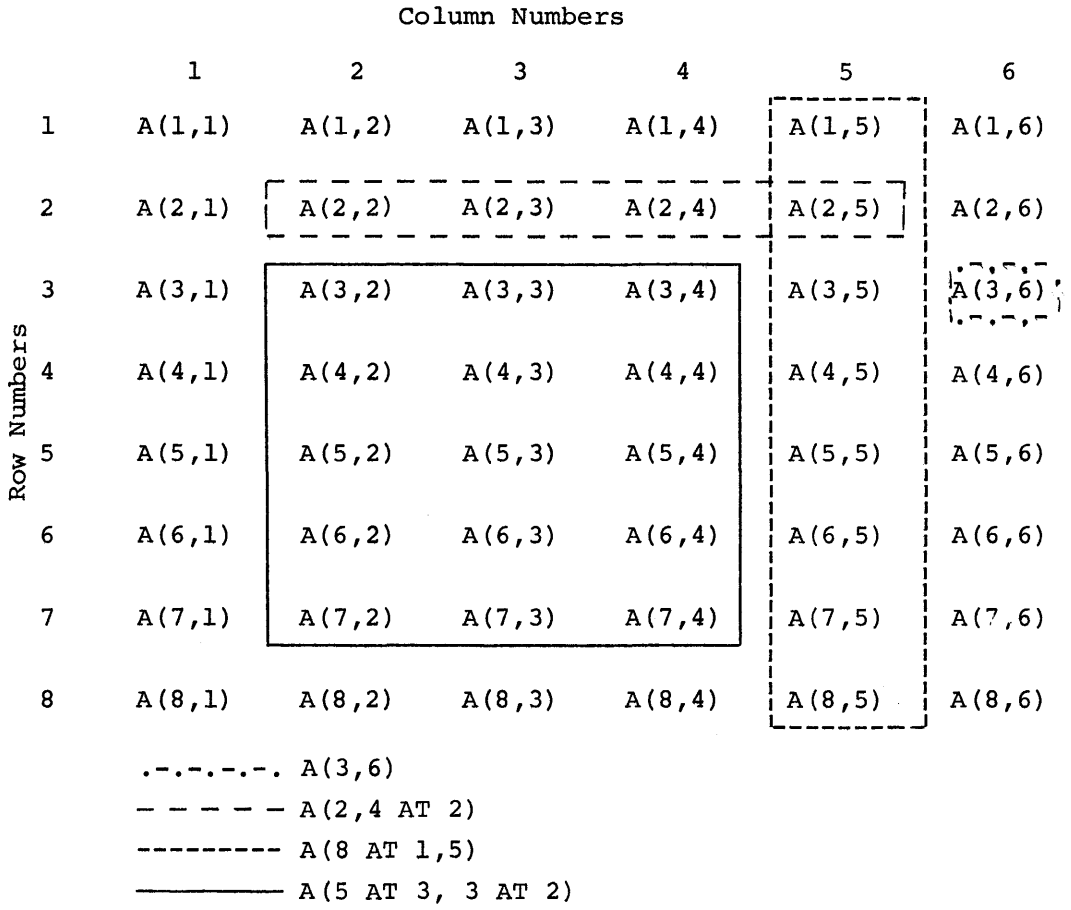


Figure 13-3

Subarray Subscripting of a Two-Dimensional Array

An analogous interpretation applies to subarray subscripting in the first position only:

```
A(8 AT 1,5)
```

Again, we have an array of rank 1; this one has size 8. In the diagram it is represented by elements of a column -- all of them, in this case, since there are just 8 rows.

In the final case we have two subarray subscripts:

```
A(5 AT 3,3 AT 2)
```

This names an array of rank 2, with first and second dimension sizes 5 and 3, respectively. It thus represents a total of 15 elements, which correspond to all A(I,J) with $3 \le I \le 7$ and $2 \le I \le 4$. In the diagram, this array is represented by a smaller rectangle, formed by the intersection of rows 3 through 7, and columns 2 through 4.

We can generalize our observations so far in the following table:

expression	rank	size of 1st dimension	size of 2nd dimension	elements represented
A(I1,J1)	0	---	---	A(I1,J1) only
A(I1,J1 AT J2)	1	J1	---	A(I1,J), J2≤J≤J2+J1-1
A(I1 AT I2,J1)	1	I1	---	A(I,J1), I2≤I≤I2+I1-1
A(I1 AT I2, J1 AT J2)	2	I1	J1	A(I,J), I2≤I≤I2+I1-1, J2≤J≤J2+J1-1

In each case, the rank of the expression is equal to the number of subarray subscripts; the size of the first dimension is equal to the size of the first subarray subscript, if any (by the size of a subarray subscript we mean the value preceding the word AT); and the size of the second dimension is equal to the size of the second subarray subscript, if any. The elements represented are exactly those which "match" the subscript list in both places -- by being equal to a single subscript, or within the range specified by a subarray subscript. (Notice that, as in the example A(I1,J1 AT J2), the first subscript may actually be the second subscript in the list. Thus A(I1,J1 AT J2) has the same rank and size as A(J1 AT J2,I1), though their meanings are quite different -- the former represents a "column", the latter a "row".)

It is only a short step to generalizing the subarray subscripting rules to arrays of any number of dimensions. Given any subscripted variable -- the name of an n-dimensional array followed by a list of n subscripts, single or subarray -- we can infer:

1) It represents a value whose rank is equal to the number of subarray subscripts in the list.

2) The size of its kth dimension is equal to the size of the kth subarray subscript in the list.

3) It represents exactly those elements which "match" the list of subscripts in every place.

To see if you understand this definition, try to follow it through for the array

```
VARIABLE B IS ARRAY(6,2,8) REAL;
```

subscripted

```
B(3 AT 4,1,4 AT 5)
```

You should be able to see why this is a subarray of rank 2, with 1st dimension size 3 and 2nd dimension size 4, representing the following 12 elements of B:

```
B(4,1,5)    B(4,1,6)    B(4,1,7)    B(4,1,8)
B(5,1,5)    B(5,1,6)    B(5,1,7)    B(5,1,8)
B(6,1,5)    B(6,1,6)    B(6,1,7)    B(6,1,8)
```

(In what order are these elements printed by

```
PRINT (B(3 AT 4,1,4 AT 5)); ?
```

According to the unraveling rules, the second subscript of a rank 2 array is varied faster than the first. So the elements are printed in the order you get by reading the first row above from left to right, followed by second row from left to right, followed by the third from left to right.)

A shorter notation for certain subarrays

In many applications a subarray subscript is used to select all elements of a particular dimension. For instance, in the last section we showed how a 2-dimensional array:

```
VARIABLE A IS ARRAY(8,6) INTEGER;
```

can be subscripted

```
A(8 AT 1,5)
```

to represent a "slice" through the entire array -- that is, an array of all elements of A with second-subscript 5.

CS-4 provides a convenient, shorter notation for slices of this sort: the entire subarray subscript can be replaced by the single character *. Thus A(8 AT 1,5) can be abbreviated

```
A(*,5)
```

Similarly, we can use * in the second dimension:

```
A(I,*)
```

as an abbreviation, in this case, for

```
A(I,6 AT 1)
```

A(*,*), it should be clear, represents the same set of values as A.

The use of * is not limited to two-dimensional arrays. For instance, given

```
VARIABLE B IS ARRAY(5,2,8) REAL;
```

then the following pairs of expressions have identical meanings:

B(3,2,*)	B(3,2,8 AT 1)
B(3,*,4)	B(3,2 AT 1,4)
B(*,1,*)	B(5 AT 1,1,8 AT 1)
B(*,*,7)	B(5 AT 1,2 AT 1,7)
B(*,*,*)	B(5 AT 1,2 AT 1,8 AT 1) or B

Distributive operators

Operators which distribute over single-dimensional arrays are also distributive over arrays of rank 2, 3, or 4. The rules we have already given are easily extended to cover the multi-dimensional cases -- just substitute the word "arrayness" for "size". Thus when an operator takes two array operands (of any rank from 1 to 4), they must have the same arrayness; and the result of a distributed operation has the same arrayness as the operand(s). The rules governing legal element modes, and the actual distribution of the operations, are as before.

When you distribute an infix operator between two subarrays, it's important to keep straight just which elements of the two arrays correspond. For instance, given

```
VARIABLE C IS ARRAY(5,5) INTEGER,
   D IS ARRAY(6,6,2) INTEGER;
```

it is legal to write

```
C(3 AT 1,2 AT 3) * D(3 AT 2,5,*)
```

since both operands have rank 2, 1st dimension size 3, and 2nd dimension size 2. Six multiplications will be performed:

```
C(1,3) * D(2,5,1)
C(1,4) * D(2,5,2)
C(2,3) * D(3,5,1)
C(2,4) * D(3,5,2)
C(3,3) * D(4,5,1)
C(3,4) * D(4,5,2)
```

ALL and ANY

The array-handling expressions using ALL and ANY -- introduced in Chapter 12 -- may be applied to BOOLEAN arrays of any rank. Their value is always computed by applying the appropriate operator -- & for ALL, | for ANY -- between all the elements of the array being operated upon. Thus for example if one declares

```
VARIABLE TABLE IS ARRAY(10,12,2) BOOLEAN;
```

then the expression

```
ALL (TABLE)
```

is a single BOOLEAN which is true if and only if all 240 elements of TABLE are TRUE.

(By the way, the SUM and PRODUCT expressions of Chapter 11 cannot be applied to multi-dimensional arrays of INTEGERs or REALs, except for certain special ones of modes we have not yet introduced.)

Using the array-handling tools

Making use of the material in the last few sections, we can now rewrite the temperature-grid program in more compact form:

```
VARIABLE TEMPERATURE IS ARRAY(9,6) REAL ::= 0,
   NEW_TEMPERATURE IS ARRAY(7,4) REAL,
   EQUILIBRIUM IS BOOLEAN ::= FALSE;
READ_LINE (TEMPERATURE(7 AT 2,1));
READ_LINE (TEMPERATURE(7 AT 2,6));
READ_LINE (TEMPERATURE(1,4 AT 2));
READ_LINE (TEMPERATURE(9,4 AT 2));
UNTIL EQUILIBRIUM REPEAT
   REPEAT
      NEW_TEMPERATURE := (TEMPERATURE(7 AT 1, 4 AT 2)
         + TEMPERATURE(7 AT 3,4 AT 2)
         + TEMPERATURE(7 AT 2,4 AT 1)
         + TEMPERATURE(7 AT 2,4 AT 3)) / 4;
      IF ANY (NEW_TEMPERATURE - TEMPERATURE(7 AT 2,4 AT 2)
         > 0.001) |
         ANY (NEW_TEMPERATURE - TEMPERATURE(7 AT 2,4 AT 2)
         < -0.001)
         THEN TEMPERATURE(7 AT 2,4 AT 2) := NEW_TEMPERATURE;
      ELSE EQUILIBRIUM := TRUE;
   END;
```

```
TEMPERATURE(1,1) := (TEMPERATURE(1,2) + TEMPERATURE(2,1)) / 2;
TEMPERATURE(1,6) := (TEMPERATURE(1,5) + TEMPERATURE(2,6)) / 2;
TEMPERATURE(9,1) := (TEMPERATURE(8,1) + TEMPERATURE(9,2)) / 2;
TEMPERATURE(9,6) := (TEMPERATURE(9,5) + TEMPERATURE(8,6)) / 2;
PRINT (TEMPERATURE);
```

Every REPEAT loop that did array processing in the old program is here replaced with a whole-array expression. Note that NEW_TEMPERATURE can be conveniently declared as an ARRAY(7,4) REAL -- giving it the same arrayness as TEMPERATURE(7 AT 2,4 AT 2), which represents the interior of the grid. Also, by the rules we gave, the elements of TEMPERATURE are printed out in the same order as before.

Arrays of arrays

Our basic exposition of multi-dimensional arrays is now complete. In the next few sections we present another programming problem, which serves to show the need for multi-dimensional arrays from a different point of view.

The problem is this: we want to change the center-of-gravity program of previous chapters so that it finds the center of gravity of a group of stationary weights, positioned anywhere in three-dimensional space -- not just along a one-dimensional rod. For simplicity, we assume -- as in Chapter 10 -- that each case has input for exactly five weights.

In the one-dimensional rod problem, as you may remember, the program depends on the following formula:

$$C = \sum_{i=1}^{5} w_i x_i \; / \; \sum_{i=1}^{5} w_i$$

The position of each weight value w_n is specified by the corresponding value x_n, which is the weight's distance, positive or negative, from the rod's center point. Only a single x_i value is needed to determine each weight's position, because the rod is one-dimensional. For the same reason, only one value is needed to specify the position of the center of gravity, C.

In the three-dimensional problem, we use Cartesian coordinates to fix positions. That is, we imagine a set of three "rods", called axes, each perpendicular to the other two, and all three intersecting at a common center point. Now, each w_n's position must be specified by <u>three</u> distance values -- call them x_{n1}, x_{n2}, and x_{n3}. The value x_{n1} is weight w_n's distance from the center point along the first axis, x_{n2} is its distance along the second axis, and x_{n3} is its distance along the third axis. In the same way, the position of the center of gravity is specified by three values: C_1, C_2, and C_3. So in the three-dimensional problem, we have a set of three formulas, one for each C_j:

$$C_j = \sum_{i=1}^{5} w_i x_{ij} \; / \; \sum_{i=1}^{5} w_i \qquad 1 \le j \le 3$$

Notice that each C_j is computed from a different sum of products.

The one-dimensional rod program of Chapter 10, to refresh your memory, looked like this:

```
VARIABLES WEIGHT, DISTANCE ARE ARRAY(5) REAL,
   WEIGHT_SUM, PRODUCT_SUM, CENTER ARE REAL,
   I IS INTEGER;
REPEAT
   WEIGHT_SUM := 0;
   PRODUCT_SUM := 0;
   READ_LINE (WEIGHT, DISTANCE);
   FOR I THRU 5
      REPEAT
         WEIGHT_SUM := WEIGHT_SUM + WEIGHT(I);
         PRODUCT_SUM :=
            PRODUCT_SUM + WEIGHT(I) * DISTANCE(I);
      END;
   IF WEIGHT_SUM = 0 THEN TERMINATE;
   CENTER := PRODUCT_SUM / WEIGHT_SUM;
   PRINT (WEIGHT, DISTANCE, CENTER);
END;
```

In a three-dimensional program, CENTER must represent a list of three values (C_1, C_2, and C_3), and for each CENTER value there is a different PRODUCT_SUM; so CENTER and PRODUCT_SUM are declared as arrays of three REALs:

```
VARIABLES CENTER, PRODUCT_SUM ARE ARRAY(3) REAL;
```

Each element of DISTANCE should also represent three values (x_{n1}, x_{n2}, and x_{n3}). That is, we want each element of DISTANCE to be, not a single REAL, but an ARRAY(3) REAL; we would like to be able to declare DISTANCE as

```
VARIABLE DISTANCE IS ARRAY(5) ARRAY(3) REAL;
```

Given only these new declarations, we wouldn't have to make any other changes to handle the three-dimensional weight problem! The statement

```
READ_LINE (WEIGHT, DISTANCE);
```

would read in a REAL value for each element of WEIGHT, and three REAL values for each element of DISTANCE. The assignment in the inner loop:

```
PRODUCT_SUM := PRODUCT_SUM + WEIGHT(I) * DISTANCE(I);
```

calculates all three sums of products simultaneously, using the distributivity of + and * and array assignment. Similarly,

```
CENTER := PRODUCT_SUM / SUM;
```

calculates all three values of CENTER using array division and assignment.

The meaning of arrays of arrays

We have shown a simple and elegant way to adapt the weights program to three dimensions, by changing nothing but a few declarations. But two questions remain: Is the declaration

```
VARIABLE DISTANCE IS ARRAY(5) ARRAY(3) REAL;
```

a legal CS-4 statement? And if so, what does it tell us about DISTANCE -- in particular, what are its rank and dimension sizes?

If you can remember back to what we said about the ARRAY mode generator in Chapter 10, you will see that the declaration of DISTANCE has to be legal. We said then that a generator like ARRAY(5) could be followed by any mode name; and we said that a generated mode like ARRAY(3) REAL is as valid a mode name as any other. Putting these principles together, we have to allow

```
ARRAY(5) ARRAY(3) REAL
```

in CS-4.

Passing to the second question, we can say this: DISTANCE has precisely the same arrayness as a variable of mode

```
ARRAY(5,3) REAL
```

DISTANCE has a total of 5 * 3 = 15 REAL elements; its first dimension has size 5, and its second dimension has size 3. Subscripting DISTANCE with a single value:

```
DISTANCE(I)        1 ≤ I ≤ 5
```

yields a slice of mode ARRAY(3) REAL -- a one-dimensional slice of size 3. Subscripting DISTANCE(I) with a second value:

```
DISTANCE(I)(J)     1 ≤ I ≤ 5, 1 ≤ J ≤ 3
```

yields a single REAL value.

All these similarities derive from one basic fact: the two declarations

```
VARIABLE DISTANCE IS ARRAY(5) ARRAY(3) REAL;
VARIABLE DISTANCE IS ARRAY(5,3) REAL;
```

are entirely equivalent in CS-4. Either declaration makes the new weights program work. As a consequence of this equivalence, the expressions DISTANCE(I)(J) and DISTANCE(I,J) have the same meaning; so do the slice expressions DISTANCE(I) and DISTANCE(I,*).

Furthermore, as you would expect, the properties revealed by this one example can be generalized to arrays of any rank:

1) A multi-dimensional array may optionally be defined in terms of arrays of arrays. For example, the following declare identical arrays named C:

```
VARIABLE C IS ARRAY(5,6,7) INTEGER
VARIABLE C IS ARRAY(5,6) ARRAY(7) INTEGER;
VARIABLE C IS ARRAY(5) ARRAY(6,7) INTEGER;
VARIABLE C IS ARRAY(5) ARRAY(6) ARRAY(7) INTEGER;
```

2) Single subscripts may be broken up for clarity, if you desire. A(1,2,3), for instance, can be written A(1)(2)(3) or A(1,2)(3) or A(1)(2,3).

3) Subscript *'s at the end of a subscript list are optional; so for instance if B is a rank 4 array, the following pairs mean the same:

B(1)	B(1,*,*,*)
B(4 AT 2,4)	B(4 AT 2,4,*,*)
B(3,*,5)	B(3,*,5,*)

Multiple subscripts in general

We have said that when the subscripts on an array are all single, not sub-array, they may be combined into one subscript list. But when some of the subscripts are sub-array, combining subscripts is no longer always valid.

To see why, assume we have this declaration from the previous section:

```
VARIABLE DISTANCE IS ARRAY(5) ARRAY(3) REAL;
```

Remember that we explained the meaning of DISTANCE(2)(3) by saying that it represented the 3rd element of the rank 1 array DISTANCE(2); this turned out to be just the same thing as DISTANCE(2,3). But now consider this expression:

```
DISTANCE(4 AT 2)(3)
```

By the same reasoning, this must be interpreted as the third element of the rank 2 array DISTANCE(4 AT 2). Thus it is a rank 1 array of 3 elements which is the same as DISTANCE(4,*). On the other hand,

```
DISTANCE(4 AT 2,3)
```

though it is also of rank 1, represents a different set of four elements.

In general, a subscript is applied to whatever value is written to the left of it. When there are two or more subscripts on a variable, this means that each one is applied, independently, to whatever array results from applying all the preceding subscripts. Subscripts applied independently do not always produce the same value that they would if applied together, in a comma-separated list.

There is no limit to the number of sub-array subscripts that may be applied independently to a variable, provided they are all within bounds. One could conceivably write:

```
TEMPERATURE(7 AT 2,4 AT 2)(5 AT 2)(2 AT 2,3 AT 2)
```

which represents the same array as TEMPERATURE(2 AT 4,3 AT 3). Any single subscript, however, reduces the rank of the result by 1; so the number of single subscripts applied to an array may never exceed its rank.

14.0

CHARACTER STRING DATA

This chapter is an introduction to the use of character strings as data in CS-4. It shows how to include messages in output, to clarify it or make it easier to read. It also explains how to declare variables whose values are character strings, so that words or names can be read from input cards, stored, and output later with the computed results.

Computers actually do many more things with character strings than just input, store, and output them. The compiler, for instance, is a huge program that accepts character strings -- CS-4 programs -- as input; it performs a complicated analysis on these strings to determine their meaning and translate them. In the process of translation it also constructs new character strings, which are later printed to form the tables, error messages, and so on that you read on the compilation listing. This chapter only begins the discussion of string processing; we will continue to look at strings, and the tools for working with them, in the following chapter.

Character sets

A character string, or string, is a piece of data whose value is, as the name implies, a sequence of characters. In some ways, strings are similar to other modes -- INTEGERs, REALs, BOOLEANs -- which represent single pieces of data. But since strings are a different class of values -- character-sequence values rather than numerical or truth values -- they have different properties and are treated somewhat differently in CS-4.

Just what is a "character"? The most familiar ones are those associated with what are called printing graphics -- letters, numerals, symbols used in operators, decimal point, semicolon, comma, parentheses, and others. Also included is the space character -- which shows up as a blank space when printed. Certain other characters, called control characters, also have no associated printing graphic; they are used for various purposes, such as sending instructions to printers and other output devices. (For the most part, control characters are inserted automatically by statements such as PRINT, and you need not be concerned about them.)

The entire group of characters available to make up strings is called the character set. Different users have different character sets available to them, depending on the input and output devices they employ; the installation where you run your programs can tell you what character sets are available to you. (To be consistent, this primer uses the ASCII character set throughout, as does the Language Reference Manual. A complete list of ASCII printing graphics appears in an appendix to the present volume.)

String literals

You can refer to a particular string value in a program by writing a string literal -- just as you refer to particular INTEGER, REAL or BOOLEAN values by means of literals. A string literal is written as a sequence of characters surrounded by apostrophe characters:

```
'THE CENTER OF GRAVITY IS:'
'NO. OF INPUT VALUES EXCEEDS MAXIMUM OF 5.'
'INTEREST PAYABLE ='
'   OLD BALANCE   '
```

The value of each of these literals is precisely the string of characters between the apostrophes. If a program contained the statements

```
PRINT ('THE CENTER OF GRAVITY IS:');
PRINT (CENTER);
```

and CENTER had the value 3.0, the following lines would be printed out:

```
THE CENTER OF GRAVITY IS:
3.00000E+00
```

Note that the apostrophes are not part of the string represented by the string literal; they just mark the beginning and end of the literal.

All spaces within a string literal are significant. Each one represents a space character in the character string. The importance of spaces in a literal can be seen when it is printed out; the statement

```
PRINT ('THE VALUE OF COUNT = ', COUNT);
```

causes, when COUNT is 5, the output

```
THE VALUE OF COUNT = 5
```

while this PRINT statement:

```
PRINT ('   THE VALUE OF COUNT =   ', COUNT);
```

results in this output:

```
   THE VALUE OF COUNT =   5
```

A PRINT statement like this does not leave a space after strings the way it does after numeric values. So if you want one space between the = and the 5, leave a space between the = and the closing apostrophe of the character string. Comments, which normally are equivalent to spaces, are not recognized within string literals. A start-comment character (# or {) appearing in a string literal is interpreted as just another character in the string. So the statement

```
PRINT ('ACCOUNT #');
```

prints out the line

```
ACCOUNT #
```

as you would want it to.

A special problem arises in writing the literal for a string which itself contains an apostrophe. Consider, for instance, how to print out the following line:

```
INVALID 'SIZE' VALUE -- CAN'T EXCEED 50
```

It wouldn't do to write a PRINT statement like this:

```
PRINT ('INVALID 'SIZE' VALUE -- CAN'T EXCEED 50');
```

because the compiler will assume the apostrophe before SIZE marks the end of the string -- it is unable to tell that this apostrophe, and the two that follow, are supposed to be part of the string. To get around this problem, a special rule is provided: an apostrophe character within a string is represented by two consecutive apostrophes in the string's literal representation. The desired output, therefore, can be produced by

```
PRINT ('INVALID ''SIZE'' VALUE -- CAN''T EXCEED 50');
```

This notation is perfectly unambiguous: a single apostrophe closes a string literal, while a pair of consecutive apostrophes represents a single apostrophe within the literal.

It is sometimes useful to speak of the length of a string. By this we mean the number of characters it contains. The length of a string literal can be determined from the number of characters between the apostrophes:

string literal	length
'THE CENTER OF GRAVITY IS:'	25
' OLD BALANCE '	28
'='	1
' '	7

Keep in mind, however, that a pair of apostrophes within a literal represents only a single character in the string. So

```
'INVALID ''SIZE'' VALUE -- CAN''T EXCEED 50'
```

has a length of 39, although there are 42 characters in all between the first and last apostrophes.

Adding useful messages to output

By adding some messages to the center-of-gravity program of Chapter 12, we can make it a lot easier to tell what actually happened during execution. As the program stood in Chapter 12, there were four different conditions on which it would terminate:

(1) the array size input, N, could be out of range; (2) one or more distance values could be out of range; (3) one or more weight values could be negative; (4) all the weight values could be zero. Case (4) was the "normal" termination; the other three were errors. But there was no way to tell from the output which condition caused termination in any particular run.

Using character string literals in PRINT statements, we can produce messages that distinguish the four cases. We'll also change the operation of the program somewhat: on conditions (2) and (3), execution of the program won't be terminated. Instead, calculation of the center of gravity will be skipped for the erroneous case, and the program will proceed to the next case. WEIGHT and DISTANCE will still be printed out, so you can find the out-of-range distance or negative weight that caused the error. We'll introduce a BOOLEAN variable, SKIP_CASE, that records when condition (2) or (3) holds.

Here's one way the program can be written, with the message literals inserted:

```
VARIABLES WEIGHT, DISTANCE ARE ARRAY(25) REAL,
   WEIGHT_SUM, PRODUCT_SUM, CENTER ARE REAL,
   N IS INTEGER,
   SKIP_CASE IS BOOLEAN ::= FALSE;
REPEAT
   READ_LINE (N);
   IF N < 1 | N > 25 THEN
      BEGIN;
         PRINT('*****ERROR:  SIZE VALUE OF ', N,
            'IS OUT OF RANGE.');
         PRINT ('     SIZE MUST BE >= 1 and <= 25.');
         PRINT ('EXECUTION HAS BEEN TERMINATED DUE TO ',
            'THIS ERROR.');
         TERMINATE;
      END;
   READ_LINE (WEIGHT(N AT 1), DISTANCE(N AT 1);
   IF ALL (WEIGHT(N AT 1) = 0) THEN
      BEGIN;
         PRINT ('EXECUTION TERMINATED NORMALLY ON CASE ',
            'WITH ALL WEIGHTS = 0.');
         TERMINATE;
      END;
```

```
      IF ANY (WEIGHT(N AT 1) < 0) THEN
         BEGIN;
            PRINT ('*****ERROR:  ONE OR MORE WEIGHT VALUES ',
               'ARE NEGATIVE.');
            SKIP_CASE := TRUE;
         END;
      IF ANY (DISTANCE(N AT 1) > 5 | DISTANCE(N AT 1)
         < -5) THEN
         BEGIN;
            PRINT ('*****ERROR:  ONE OR MORE DISTANCE VALUES ',
               'ARE OUT OF RANGE.');
            PRINT ('     DISTANCE VALUES MUST BE <= 5 AND ',
               '>= -5.');
            SKIP_CASE := TRUE;
         END;
      PRINT ('NUMBER OF WEIGHTS IS ', N);
      PRINT ('WEIGHT VALUES ARE:');
      PRINT (WEIGHT(N AT 1));
      PRINT ('DISTANCE VALUES ARE:');
      PRINT (DISTANCE(N AT 1));
      IF ~SKIP_CASE THEN
         BEGIN;
            WEIGHT_SUM := SUM (WEIGHT(N AT 1));
            PRODUCT_SUM :=
               SUM (WEIGHT(N AT 1) * DISTANCE(N AT 1));
            CENTER := PRODUCT_SUM / WEIGHT_SUM;
            PRINT ('POSITION OF CENTER OF GRAVITY IS ',
               CENTER);
         END;
      ELSE
         BEGIN;
            PRINT ('CENTER OF GRAVITY NOT CALCULATED FOR ',
               'THIS CASE DUE TO ERROR.');
            SKIP_CASE := FALSE;
         END;
      PRINT;
   END;
```

(Note how we've had to break some of the longer character strings in two. It's tempting to write simply

```
PRINT ('CENTER OF GRAVITY NOT CALCULATED FOR
   THIS CASE DUE TO ERROR.');
```

But the literal in this PRINT starts on one card and ends on the next; it includes all the spaces after FOR on the first card, and all the spaces preceding THIS on the second. When we break the literal in two, these unwanted spaces are not included, but the indentation of the program is maintained.)

Here's an example of what the program does. Suppose it has the following input cards to read:

```
4
10.3      13.5      7.2       9.1
-4.0      -1.1      3.1       3.9
2
5.5       -.1
2.5       4.9
5
12.2      14.7      0.0       19.0      2.2
-4.9      -0.3      1.1       4.9       5.1
27
10.0      10.0
-2.0      3.0
1
0.0
0.0
```

There are two keypunching errors: -.1 has been typed in the fifth line instead of 0.1, and 27 was punched in the tenth line when 2 was intended. The user of the program has also made the mistake of specifying a distance value of 5.1 in the ninth line. Our revised program provides the following output:

```
NUMBER OF WEIGHTS IS 4
WEIGHT VALUES ARE:
1.03000E+01 1.35000E+01 7.20000E+09 9.10000E+00
DISTANCE VALUES ARE:
-4.00000E+00 -1.10000E+00 3.10000E+00 3.90000E+00
POSITION OF CENTER OF GRAVITY IS 4.38903E-02

*****ERROR:  ONE OR MORE WEIGHT VALUES ARE NEGATIVE.
NUMBER OF WEIGHTS IS 2
WEIGHT VALUES ARE:
5.50000E+00 -1.00000E-01
DISTANCE VALUES ARE:
2.50000E+00 4.90000E+00
CENTER OF GRAVITY NOT CALCULATED FOR THIS CASE DUE TO ERROR.

*****ERROR:  ONE OR MORE DISTANCE VALUES ARE OUT OF RANGE.
     DISTANCE VALUES MUST BE <= 5 AND >= -5.
NUMBER OF WEIGHTS IS 5
WEIGHT VALUES ARE:
1.22000+E+01 1.47000E+01 0.00000E+00 1.90000E+01
2.20000E+00
DISTANCE VALUES ARE:
-4.90000E+00 -3.00000E-01 1.10000E+00 4.90000E+00
5.10000E+00
CENTER OF GRAVITY NOT CALCULATED FOR THIS CASE DUE TO ERROR.

*****ERROR:  SIZE VALUE OF 27 IS OUT OF RANGE.
     SIZE MUST BE >= 1 AND <= 25.
EXECUTION HAS BEEN TERMINATED DUE TO THIS ERROR.
```

Notice that the last three cards are never read, because of the termination due to error.

Character string variables

To have a variable represent character-string values, it must be declared with the mode STRING:

```
VARIABLE NAME IS STRING(24);
```

The number in parentheses following STRING is the length of the strings represented by the variable. In the example, NAME is declared so that it can be assigned character strings of length 24:

```
NAME := 'DAVID G. COOPER, MANAGER';
NAME := 'BRUCE S. MARTEN         ';
NAME := '                        ';
```

The length value of a STRING-mode variable cannot be changed once it is fixed in a declaration. Thus NAME can only represent strings of length 24. However, strings shorter than 24 may appear to the right of the assignment operator; such strings' values are extended to the right with space characters before they are assigned to NAME. Hence the following all assign NAME the same value:

```
NAME := 'DAVIDOFF                ';
NAME := 'DAVIDOFF       ';
NAME := 'DAVIDOFF';
```

It is an error to assign a string longer than 24 to NAME.

Of course, the value of one STRING variable may also be assigned to another STRING variable. In general, an assignment of the form:

```
STRING1 := STRING2;
```

is valid if the length of STRING2 is less than or equal to the length of STRING1. If STRING2 is shorter than STRING1, enough spaces are added to the right of STRING2's value to make it as long as STRING1.

Like variables of other modes, STRING variables can be assigned values by READ_LINE statements, and have their values printed out by PRINT statements.

Using STRING variables

One piece of data conspicuously lacking in our sample billing program is the name of the customer who has each account. Using a STRING-mode variable, we can now read in a name with each set of input figures, and write it out with the output.

We'll say a name is at most thirty characters; so we declare

```
VARIABLE CUSTOMER_NAME IS STRING(30);
```

Supposing that the name input string is placed before the rest of the customer's data, the billing program READ_LINE statement would be changed to:

```
READ_LINE (CUSTOMER_NAME, ACCT_NO, OLD_BAL, PAYMENT
   PURCHASE);
```

The input for each case might now be on a card that looks like this:

```
MAURICE R. STANTON              705  5950  2975  9905
```

Notice that enclosing apostrophes do not appear around strings on the input card. Instead of looking for a string within apostrophes, READ_LINE simply inputs however many characters necessary to fill the string variable. In this case, since CUSTOMER_NAME is of length 30, and since it is the first variable in READ_LINE, the contents of the first 30 positions on the card (whatever they are -- blanks included) are assigned to CUSTOMER_NAME. Then, beginning in position 31, READ_LINE begins looking for the values to assign to the other variables. (Rules governing STRING variables in READ_LINE will be discussed in more detail in a later chapter.)

We could also add to the billing program a more informative set of PRINT statements:

```
PRINT ('ACCOUNT # ', ACCT_NO, '      ', CUSTOMER_NAME);
PRINT ('OLD BALANCE:      ', OLD_BAL);
PRINT ('PAYMENT REC''D:    ', PAYMENT);
IF ~INTEREST_FREE THEN
   PRINT ('BALANCE PAST DUE: ', PAST_DUE,
      '    INTEREST ON PAST DUE BAL.: ', INTEREST);
PRINT ('PURCHASES MADE:   ', PURCHASE);
PRINT ('NEW BALANCE:      ', NEW_BAL);
PRINT;
```

If account #705 is not interest-free, the output will be:

```
ACCOUNT # 705       MAURICE R. STANTON
OLD BALANCE:      5950
PAYMENT REC'D:    2975
BALANCE PAST DUE: 2975     INTEREST ON PAST DUE BAL.: 45
PURCHASES MADE:   9995
NEW BALANCE:      13015
```

If it is interest-free, the output will be one line shorter:

```
ACCOUNT # 705          MAURICE R. STANTON
OLD BALANCE:       5950
PAYMENT REC'D:     2975
PURCHASES MADE:    9995
NEW BALANCE:       12970
```

15.0

STRING PROCESSING

Now that you are familiar with the concept of character string data, we can go on to explain how strings are manipulated in a program. This chapter shows how to refer to substrings of a string, and how to put strings or substrings together to form larger strings. It also gives the definitions of comparison operators and some conversions for string data.

Strings as arrays

You may already have noticed some similarities between character strings and one-dimensional arrays. Characters are the units of a string, just like elements such as INTEGERs are the units of an array. The length of a string (the number of characters) is akin to the size of an array (number of elements). Both ARRAY and STRING variables cannot change size (or length) once their sizes are fixed in a declaration.

These correspondences are not a coincidence. Fundamentally, strings are "arrays of characters" in the same way that other variables you have learned to declare are arrays of INTEGERs. However, STRINGs have some important differences from the sort of array we have been declaring with the ARRAY mode generator. Furthermore, certain operators behave specially with STRING operands -- assignment, for instance, was shown in the last chapter to take two STRING operands of different lengths, although it cannot take two INTEGER array operands of different sizes. This chapter explores the similarities and differences in the behavior of STRINGs and other arrays.

Subscripting STRING variables

One important way in which strings are like arrays is that they can be subscripted. A single subscript selects a character from the string, while a subarray subscript selects a substring. This is just like the situation with regard to arrays, where a single subscript on an array of REALs selects a single REAL, while a subarray subscript on an array of REALs selects a subarray of REALs.

What is the mode of a "substring" from a string? It is STRING, and its length is equal to the value before AT in the subscript. In this respect, too, STRINGs behave in a manner exactly analogous to ARRAYs. If NAME is a STRING(25), then NAME(I AT J) is, as you would expect, a STRING of length I.

If you think of the nth character of a string as its nth "element", then the rules for applying subscripts to strings should be evident from your knowledge of arrays. Here's an example to make things clear: assume

```
VARIABLE NAME IS STRING(20);
NAME := 'BROWNEL, BEN';
```

then here are the values of some subscriptings of NAME:

Subscripted Variable	Value (written as a literal)
NAME(1)	'B'
NAME(8)	','
NAME(20)	' '
NAME(7 AT 1)	'BROWNEL'
NAME(3 AT 10)	'BEN'
NAME(13 AT 1)	'BROWNEL, BEN'
NAME(8 AT 13)	' '

Since a subscripted string variable is itself of STRING mode, it may be used just like an unsubscripted variable of the same length. For instance, one can write

```
PRINT (NAME(4 AT 10), NAME(7 AT 1));
```

to print out

```
BEN BROWNEL
```

A substring may have a value assigned to it; for example, either of the following:

```
NAME(4 AT 10) := 'JOHN';
NAME(11 AT 10) := 'JOHN';
```

changes the value of NAME to 'BROWNEL, JOHN '. It would be illegal, however, to write

```
NAME(4 AT 10) := 'HAROLD';
```

because the length of 'HAROLD', 6, exceeds the length of NAME(4 AT 10), which is 4.

String equality comparisons

The equality operators = and ~= may be applied to two STRING operands. Two STRING values are equal if and only if they have the same lengths, and they consist of the same characters in the same order.

Note that = and ~= act quite differently on STRINGs than they do on one-dimensional INTEGER or REAL arrays. Equality comparisons of the latter are distributive -- they are applied between each of the elements, and produce an array of BOOLEANs. But equality comparison treats STRING values as single entities, and produces a single BOOLEAN result for two STRING operands of any length.

An example with subscripted strings

In our modifications to the billing program in the last chapter, we added a string variable to store the name of the customer who held each account:

```
VARIABLE CUSTOMER_NAME IS STRING(30);
```

Every time we read in account data, we read a character string into CUSTOMER_NAME; later we printed out this string along with ACCT_NO:

```
PRINT ('ACCOUNT # ', ACCT_NO, '      ', CUSTOMER_NAME);
```

Let's now add a few extra requirements, to show some simple string processing. Assume that the strings read into CUSTOMER_NAME are like the sample one in the previous section -- last name, comma, first name:

```
BROWNEL, BEN
JOHNSON, EDMUND
MEGALOPOULOS, THEODORE
```

Suppose further that we just want to print out the last name with the account number. We must therefore search through each CUSTOMER_NAME string for a comma character, and then print out just the substring that comes before the comma.

To do this, we have to subscript CUSTOMER_NAME. The PRINT statement above is replaced by the following two statements:

```
FOR I FROM 2 THRU 30 UNTIL CUSTOMER_NAME(I) = ',' REPEAT; END;
PRINT ('ACCOUNT # ', ACCT_NO, '      ',
   CUSTOMER_NAME(I - 1 AT 1));
```

The REPEAT tests each character of CUSTOMER_NAME, from the second through the 30th, until it finds a comma. Then the last-name substring -- all the characters of CUSTOMER_NAME up to the one before the comma -- is printed out.

Concatenation

Now you know how to take strings apart, by subscripting, into characters and substrings. The next step is to learn how to put strings together, to form larger strings. This process is called concatenation.

Concatenation is an operation on two strings; it produces a single string which consists of all the characters of the first string followed by all the characters of the second. In CS-4, concatenation is performed by the infix concatenation operator, which is a pair of vertical strokes:

```
NAME || NAME2
```

Concatenation takes STRING-mode operands of any length; they may be string literals or variables, possibly subscripted.

String concatenation is a fairly intuitive operation, as a few examples should show. Again, let's assume

```
VARIABLE NAME IS STRING(20);
NAME := 'BROWNEL, BEN        ';
```

We can produce various new strings by concatenating parts of NAME:

expression	value
`NAME(10 AT 1) || '.'`	`'BROWNEL, B.'`
`'DR. ' || NAME(7 AT 1)`	`'DR. BROWNEL'`
`NAME(4 AT 10) || NAME(7 AT 1)`	`'BEN BROWNEL'`
`NAME(7 AT 1) || ' ' || NAME(12 AT 9)`	`'BROWNEL  BEN        '`
`NAME(10) || '. ' || NAME(7 AT 1)`	`'B. BROWNEL'`

Obviously, the length of A || B is equal to the length of A plus the length of B.

Concatenation is very useful for rearranging strings, as you can tell from the example. It can also serve to put related strings together. For instance, it might happen that the input to the billing program consists of two name strings, a first name and a last name, each 35 characters long; in special cases (corporate names, perhaps), the first name string might be all blank. Say that we want to print out the customer's first initial, if any, and whole last name. We would make these declarations:

```
VARIABLES FIRST_NAME, LAST_NAME ARE STRING(35),
   CUSTOMER_NAME IS STRING(38);
```

and perform this processing to create CUSTOMER_NAME:

```
READ_LINE (FIRST_NAME, LAST_NAME);
IF FIRST_NAME = '                                   ' THEN
   CUSTOMER_NAME := LAST_NAME;
ELSE
   CUSTOMER_NAME := FIRST_NAME(1) || '. ' || LAST_NAME;
```

String inequality comparisons

The four inequality operators (< <= > >=) are defined to take two STRING-mode operands. Like the equality operators, their operands need not be of equal length.

What does it mean for one string to be less than another one? If S1 and S2 consist entirely of letters, then

```
S1 < S2
```

is TRUE only when S1 comes before S2 in alphabetical order. More precisely, let N be the length of whichever of S1 and S2 is shorter. If, for some I ≤ N, S1 and S2 are the same through the I-1st character, but S1(I) comes before S2(I) in the alphabet, then S1 < S2 is TRUE; if S1(I) comes after S2(I), then S1 < S2 is FALSE. If S1 and S2 are the same through the Nth character, then the shorter of the two is considered the lesser. For example, the following expressions are all TRUE:

```
'AAAAA' < 'ZZZZZ'
'AAAAB' < 'AAAAC'
'TAAAA' < 'UAAAA'
'A' < 'ZZZ'
'AAA' < 'Z'
'BB' < 'BBB'
'HERMAN' < 'HERMANN'
```

The general definition of < is just an extension of the concept of alphabetization. The entire character set has a collating sequence -- an ordering that tells which characters come before any given character, and which come after. Thus, we define

```
S1 < S2
```

to be TRUE when S1 and S2 are the same through some (I-1)th character (I ≤ N), but S1(I) comes before S2(I) in the collating sequence; or if S1 and S2 are the same through the Nth character, and S1 is shorter than S2. Otherwise, the expression is FALSE.

Within the collating sequence for the ASCII character set (the one used in this Primer), letter characters are in alphabetical order, and digit characters are in numerical order (0 first, 9 last). The space character precedes all the digits, which precede all the letters. So, for example, we have the following TRUE relations:

```
'AAAA ' < 'AAAAA'
'AAAA1' < 'AAAAA'
'A Z' < 'AAAAA'
'A1Z' < 'AAAAA'
'12345' < '12346'
'12378' < '1238'
'234' < '2345'
```

You will find the full collating sequence for the ASCII character set in an appendix to this volume.

The meanings of the three remaining comparisons (<= > >=) are derived in the obvious way from the meanings of = and <.

It hardly needs saying that string inequalities are essential to programs that alphabetize strings. They are also valuable for certain sorts of checking; for instance, the expression

```
NAME(I) >= 'A' & NAME(I) <= 'Z'
```

is true only when NAME(I) is a letter, and so can be used to check that no spurious characters have been punched within a name.

(It is important to be aware that string inequality comparisons, like string equality comparisons, operate on strings as single entities -- they do not distribute over individual "characters". One cannot write

```
NAME > ','
```

to compare ',' with each character of NAME and so produce an array of BOOLEANs. Rather, this comparison yields a single BOOLEAN, whose value depends on the relative position of ',' and the first character of NAME in the collating sequence.)

Mixing STRING and other modes

When an INTEGER value is an argument to concatentation, it is converted to a string of digits. This makes it possible to write a statement like

```
PRINT ('ACCOUNT # ' || ACCT_NO || '.');
```

in which the literal representation of ACCT_NO is made part of a string. If ACCT_NO were 156, this line might be printed:

```
ACCOUNT # 156.
```

while if ACCT_NO were 3 the output would be

```
ACCOUNT # 3.
```

As you can see, the length of the string that ACCT_NO is converted to varies according to the number of characters needed to represent the integer. 156 converts to a string of length three, 3 converts to a string of length one, and -3 would convert to a string of length two. In the above concatenation, we left a space after # in the string 'ACCOUNT # ' so that the resulting string would have a space between # and the number.

Assignment of an INTEGER value to a STRING value also triggers conversion. For example, given

```
VARIABLE ACCT_NO IS INTEGER,
   ACCT_NO_STRING IS STRING(7);
```

then

```
ACCT_NO_STRING := ACCT_NO;
```

is legal, assuming that ACCT_NO contains seven digits or less (or a minus sign followed by six digits or less). If ACCT_NO is 156, ACCT_NO_STRING is assigned '156 '.

All the foregoing applies equally well if you substitute REAL for INTEGER. Generally, a REAL is converted to its exponential literal representation in STRING form; the length of the STRING value depends on the precision of the REAL. You can find the detailed rules spelled out in the Language Reference Manual.

There is also a conversion from BOOLEAN to STRING(5): TRUE becomes 'TRUE ', and FALSE 'FALSE'. The conversion is performed when a BOOLEAN value is an argument to a concatenation, or is assigned to a STRING variable of length 5 or more.

Arrays of strings

STRING values of a given length, like values of any mode, can be declared in arrays with the ARRAY mode generator. For example,

```
VARIABLE NAME IS ARRAY(50) STRING(20);
```

declares NAME as a one-dimensional array of 50 strings of length 20.

How is such an array of strings subscripted? Certainly NAME(I) must refer to the Ith string in NAME. But how about the Jth character of NAME(I)? That is written

```
NAME(I,J)
```

in just the same way that we subscript two-dimensional arrays. This is as it should be: a single string is subscripted as if it were an array of characters; an array of strings ought to be treated like an array of arrays of characters -- in other words, a two-dimensional array of characters. And so it is; you can even write

```
NAME(*, 10 AT 1)
```

to refer to an array of substrings of length 10, and

```
NAME(I, 10 AT 1)
```

to refer to a substring of the Ith string of NAME.

By similar reasoning, you can see that a two-dimensional array of strings:

```
VARIABLE MESSAGE IS ARRAY(2,16) STRING(24);
```

is essentially a three-dimensional array. The first two subscripts refer to "rows" and "columns" of the array, while the third subscript distinguishes particular characters of each string. And so for arrays of still higher rank: an n-dimensional array of strings is subscripted like an n+1-dimensional array.

Distributivity of string operators

The concatenation operator, and the string comparison operators, are distributive over arrays of strings. More specifically:

1) The concatenation of two arrays of strings, or of a single string and an array of strings, yields an array of strings of the same arrayness as the operand(s). If A is an array of strings of length m, and B is an array of strings of length n, then A || B is an array of strings of length m+n; similarly, if C is a single string of length k, A || C is an array of strings of length m+k.

2) A comparison of two arrays of strings, or of a single string and an array of strings, yields an array of BOOLEANs of the same arrayness as the operand(s).

Like any distributive operators, concatenation and string comparison can be applied only to two arrays of identical arrayness.

At this point you may be having some trouble distinguishing the concepts of character, character string, array of characters, and array of strings, especially when you have to figure out which ones apply when. The important distinction to keep in mind is this: for the purpose of subscripting -- to refer to substrings of various

lengths (including length 1) -- strings are treated as arrays of characters. For the purpose of operating on strings -- to concatenate or compare them -- they are treated as single, unarrayed values. This rule generalizes to arrays of strings. An n-dimensional array of strings may be treated, for the purpose of subscripting, as an n+1-dimensional array of characters. But for the purpose of operating on its string elements -- by distributing concatenation or comparison -- it is treated like an n-dimensional array of whole strings.

A string-processing example

Now we can present a fairly short program that demonstrates all the string-processing features in action. The input to this program is a single INTEGER, read into N, and N strings each 20 characters long; the program handles up to 50 input strings. Each string must contain a single name, which may be preceded or followed by blanks; there must be no blanks, or other non-letter characters, within the name.

The program reads the input strings into an array, NAME. It searches for each name among the blanks, and checks that all characters in the name are letters; if a name has blanks preceding it, they are removed (the name is moved to the beginning of the string, in other words). Then, if there were no errors, the names are printed out in two columns, in the order they were read in -- the first half of them running down the first column, and the other half running down the second.

Knowing this much, now take a look at the program:

```
VARIABLE NAME IS ARRAY(50) STRING(20),
   FIRST, LAST, I, J, N, DEPTH ARE INTEGER,
   ERROR IS BOOLEAN ::= FALSE;
READ_LINE (N);
IF N < 1 | N > 50 THEN
   BEGIN;
      PRINT ('PROGRAM CANNOT HANDLE ' || N || ' NAMES.');
      TERMINATE;
   END;
```

```
READ_LINE (NAME(N AT 1));
FOR I THRU N
   REPEAT
   NAME_SEARCH:
      BEGIN;
         FOR FIRST THRU 20 UNTIL NAME(I,FIRST) ~= ' ' REPEAT; END;
         IF NAME(I, FIRST) = ' ' THEN
            BEGIN;
               PRINT ('NAME # ' || I || ' IS BLANK.');
               ERROR := TRUE;
               EXIT FROM NAME_SEARCH;
            END;
         FOR LAST FROM 20 BY -1 UNTIL NAME(I,LAST) ~= ' '
            REPEAT; END;
         FOR J FROM FIRST THRU LAST
            REPEAT
               IF NAME(I,J) < 'A' | NAME(I,J) > 'Z' THEN
                  BEGIN;
                     PRINT ('NAME # ' || I || ' HAS INVALID' ||
                        ' CHARACTER OR EMBEDDED SPACE:');
                     PRINT (NAME(I));
                     ERROR := TRUE;
                     EXIT FROM NAME_SEARCH:
                  END;
            END; # OF REPEAT FOR J #
         IF FIRST ~= 1 THEN
            NAME(I) := NAME(I, LAST - FIRST + 1 AT FIRST);
      END NAME_SEARCH;
   END; # OF REPEAT FOR I #
IF ERROR THEN TERMINATE;
DEPTH := N / 2;
IF N ~= DEPTH * 2 THEN NAME(N + 1) := ' ';
PRINT (NAME(DEPTH AT 1) || '     ' ||
   NAME(DEPTH AT DEPTH + 1));
```

A few explanations are in order. The large REPEAT in the middle of the program does the checking of each string. The THEN clause at the end of the loop:

```
NAME(I) := NAME(I,LAST - FIRST + 1 AT FIRST);
```

moves the name to the far left of the string if it isn't already there.

DEPTH -- the number of names in each output column -- is half of N. If N is odd, DEPTH will be rounded up to the next integral value; and the IF that follows:

```
IF N ~= DEPTH * 2 THEN NAME(N + 1) := ' ';
```

adds an extra blank name at the end, so that the PRINT statement will work out properly.

The PRINT uses distributed concatenation. The expression printed:

```
NAME(DEPTH AT 1) || '      ' ||
   NAME(DEPTH AT DEPTH + 1)
```

is an array of DEPTH elements of length 45; when each element is printed out on a separate line, the result is the two-column arrangement we desire.

You can make this into a more useful program by adding statements to alphabetize the names before they are printed out. You might also want to try to adapt the billing program in a similar way, so that it stores all the names and account numbers as they are processed, and prints out at the end of the run a summary (in alphabetical order) of customers serviced.

PART 3

PROGRAMMING WITH FUNCTIONS

16.0

FUNCTIONS

In Part 1 of this Primer we demonstrated most of the basic tools for directing flow of control in CS-4 programs. At the same time, we described the manipulation and storage of the most basic types of data. Part 2 was given over to the introduction of new and more powerful tools for data handling -- arrays, array expressions, and strings.

In this and subsequent chapters, we return to the topic of flow of control. Our subject is the definition of functions -- a tool which is essential to the logical and efficient construction of all but the simplest programs.

The need for a structured program

Back in Chapter 3 we introduced a very simple example of a program that computed billing information for an imaginary set of charge accounts. In subsequent chapters we used many features of CS-4 to make that program more powerful. In one example or another, we showed how to achieve all of the following:

1) Construction of a look-up table of interest-free accounts.

2) Reading in, processing, and writing a name character string for each customer.

3) Printing out, at the end, the total number of accounts processed, and the total number of accounts with past-due balances.

4) Printing out just the account number and skipping the rest of the calculations when all the other output values would be zero.

Each of these features adds something else besides power. Each adds to the length and complexity of the program. So far, we've only incorporated one or two of these features at a time, to keep the examples reasonably short. Now let's consider implementing all of them together.

We can do this with just the tools you've learned so far -- REPEAT loops, BEGIN blocks, EXITs, and various data-handling devices. The resulting program is long and complicated indeed, compared to previous examples:

```
VARIABLES ACCT_NO, OLD_BAL, PAYMENT, PURCHASE, INTEREST,
   PAST_DUE, NEW_BAL, FREE_ACCT, FIRST, I, LAST ARE INTEGER,
   COUNT, PAST_COUNT ARE INTEGER ::= 0;
VARIABLE ERROR_MESSAGE IS STRING(40),
   NAME IS STRING(20);
VARIABLE ERROR IS BOOLEAN ::= FALSE,
   FREE_ACCT_TABLE IS ARRAY(999) BOOLEAN ::= FALSE;
                          # STORE FREE ACCT NUMBERS #
READ_LINE (N);
   IF N < 0 | N > 999 THEN
      BEGIN;
         ERROR := TRUE;
         ERROR_MESSAGE := 'NUMBER OF FREE ACCOUNTS NEGATIVE' ||
           ' OR >999';
      END;
   FOR I THRU N WHILE ~ERROR
      REPEAT
      NEXT_NUMBER:
         BEGIN;
            READ_LINE (FREE_ACCT);
            IF FREE_ACCT < 1 | FREE_ACCT > 999 THEN
               BEGIN;
                  ERROR := TRUE;
                  ERROR_MESSAGE := 'FREE ACCOUNT NUMBER' ||
                    ' IS <1 OR >999';
```

```
                EXIT FROM NEXT_NUMBER;
            END;
         FREE_ACCT_TABLE(FREE_ACCT) := TRUE;
      END NEXT_NUMBER;
   END;
                                    # MAIN PROCESSING LOOP #
FOR COUNT WHILE ~ERROR UNTIL ACCT_NO <= 0
   REPEAT
   NEW ACCT:
      BEGIN;
                                    # PROCESS NEXT ACCOUNT #
         READ_LINE (NAME, ACCT_NO, OLD_BAL, PAYMENT, PURCHASE);
                                      # PROCESS INPUT NAME #
         NAME_SEARCH:
            BEGIN;
               FOR FIRST THRU 20 UNTIL NAME(FIRST) ~= ' '
                  REPEAT; END;
               IF NAME(FIRST) = ' ' THEN EXIT FROM NAME_SEARCH;
               FOR LAST FROM 20 BY -1 UNTIL NAME(LAST) ~= ' '
                  REPEAT; END;
               FOR I FROM FIRST THRU LAST
                  REPEAT
                     IF (NAME(I) < 'A' | NAME(I) > 'Z') &
                        NAME(I) ~= ' ' & NAME(I) ~= '''' THEN
                        BEGIN;
                           ERROR := TRUE;
                           ERROR_MESSAGE := 'INVALID CHARACTER'
                              || ' WITHIN A NAME;
                           EXIT FROM NEW_ACCT;
                        END;
                  END;
```

```
            IF FIRST ~= 1 THEN
               NAME := NAME(LAST - FIRST + 1 AT FIRST);
         END NAME_SEARCH;
                                  # PROCESS INPUT DATA #
      IF ACCT_NO <= 0 THEN EXIT FROM NEW_ACCT;
      IF ACCT_NO > 999 THEN
         BEGIN;
            ERROR := TRUE;
            ERROR_MESSAGE := 'ACCOUNT HAS NUMBER >999';
            EXIT FROM NEW_ACCT;
         END;
      IF OLD_BAL = 0 & PAYMENT = 0 & PURCHASE = 0 THEN
         BEGIN;
            PRINT (ACCT_NO, NAME);
            EXIT FROM NEW_ACCT;
         END;
      IF FREE_ACCT_TABLE(ACCT_NO) | PAYMENT >= OLD_BAL THEN
         BEGIN;
            PAST_DUE := 0;
            INTEREST := 0;
         END;
      ELSE
         BEGIN;
            PAST_DUE := OLD_BAL - PAYMENT;
            INTEREST := PAST_DUE * 0.015;
            PAST_COUNT := PAST_COUNT + 1;
         END;
      NEW_BAL := OLD_BAL - PAYMENT + INTEREST + PURCHASE;
      PRINT (ACCT_NO, NAME, OLD_BAL, PAYMENT,
         PAST_DUE, INTEREST, PURCHASE, NEW_BAL);
   END NEW_ACCT;
END;
IF ERROR THEN

                              # TERMINATE WITH ERROR#
   BEGIN;
      IF COUNT = 0 THEN
         PRINT ('EXECUTION TERMINATED BY ERROR BEFORE ' ||
            'PROCESSING: ', ERROR_MESSAGE);
```

```
      ELSE
         PRINT ('EXECUTION TERMINATED BY ERROR DURING' ||
            ' PROCESSING: ', ERROR_MESSAGE);
         TERMINATE;
   END;
                                          # PRINT STATISTICS #
PRINT; PRINT;
PRINT (COUNT || ' ACCOUNTS WERE PROCESSED');
PRINT (PAST_COUNT || ' ACCOUNTS HAD BALANCES PAST DUE');
```

We've added a few comments to try to show where the different tasks of the program begin and end. Most of the routines should be pretty familiar to you by now. The input customer names are assumed to be last names only; a name must be made up of letters, spaces, and apostrophe characters. Blank name fields are allowed. We have introduced a BOOLEAN variable ERROR, which is set to **TRUE** whenever there is an error that causes termination of processing. When such an error occurs, the STRING variable ERROR_MESSAGE is assigned a message string which is printed out later in the program.

Is there any way we can make this program much shorter? Not really. What we can do is to structure it into a number of functions, so that the flow of control is clearer, and so that the program is easier to read, write, and debug.

Function calls

An identifier followed by a list of one or more values in parentheses is a fairly common expression in CS-4. Some of these expressions are subscripted array or string variables. But there are some, such as

```
PRINT (ACCT_NO, NAME);
SUM (WEIGHT);
```

which fall in another category. They are known as function calls. Each performs a specific purpose, or "function", in a program -- such as adding the elements of an array, or printing out certain values.

Let's consider the function call

```
SUM (WEIGHT)
```

in more detail. The identifier in front -- SUM -- is called the function name. The array value in parentheses is called the function's argument. (Some functions, like PRINT, may have more than one argument, in which case the arguments are written in a comma-separated list.) Each function name indicates the action, or function, to be performed, while the arguments supply values or variables that are used in performing the action. In our example, SUM indicates that array summing is to be done, while the argument tells which array is to have its elements summed.

Thus, every time SUM (WEIGHT) is encountered during execution, the elements of WEIGHT are added together. After the summing is completed, one more action is performed: a return value -- the REAL value equal to the sum of the elements -- is given to the expression SUM (WEIGHT). The function call can therefore be used as one element of a larger expression, such as

```
WEIGHT_SUM := SUM (WEIGHT);
```

which assigns the sum of the weights to WEIGHT_SUM. (There's a meaning behind the term "return value" which will become relevant later.)

How is SUM useful? It is not necessary in writing a program -- our first several array-handling examples did without it. But it does contribute to the clarity and ease of programming. Calling SUM lets you avoid the details of writing a REPEAT loop to do the summing, declaring its FOR variable, and initializing the variable that holds the partial sums. All this is taken care of by the execution of the function call.

The function name SUM is part of CS-4; that is, a SUM function call (or "call to SUM") results in the performance of a particular action which is defined as part of the language. SUM is thus called a pre-defined function. The functions PRODUCT, ALL, ANY, READ_LINE, and PRINT are also pre-defined. These functions are available to any programmer who finds their use convenient.

In this and subsequent chapters, we will show how it can be equally convenient to call functions of your own devising, in addition to using the pre-defined ones. Once you have decided what you want your functions to do, you will be able to write calls to them -- like you do with SUM -- without worrying about their internal structure. But

since these functions aren't pre-defined, you will also have to write your own definitions for them, and submit the definitions to the compiler along with the rest of your program. Writing such definitions will be the key to improving the long billing program we just displayed.

Although there are many sophisticated types of functions in CS-4, in this chapter we'll be concerned with only the very simplest sort. Our functions will not take any arguments -- so the parentheses following their names won't be written at all. Our functions won't return any values, either -- so we'll use them by themselves in statements, like we do PRINT and READ_LINE. You've already seen a few function calls of this elementary sort, such as the call to PRINT:

```
PRINT;
```

which leaves a blank line. (To be accurate, we should add that PRINT and READ_LINE do have return values. However, at this point we have no use for their return values, and so we call them as if they have none.)

Rewriting a program with function calls

Return now to the billing program. Imagine we can invent any function we want in order to simplify the program -- as long as it's a function of the very simple type we just described (no arguments, no return value). Here are some functions we could want:

STORE_FREE_ACCT_NUMBERS -- takes care of reading in the numbers of the interest-free accounts, and setting up FREE_ACCT_TABLE.

PROCESS_NEXT_ACCOUNT -- reads in the next account, and does all the account calculations, including the processing of NAME.

TERMINATE_WITH_ERROR -- prints out the appropriate error message and terminates the program.

PRINT_STATISTICS -- prints out the run statistics at the end.

Each of these functions does a different task of the program. In fact, if you look back at the comments we used to mark off the different tasks, you'll see that each function name corresponds to one of the comments.

Suppose that all the hypothetical functions could be defined to do just what we want them to do. Then we could shorten the program a great deal. The declarations would remain as they are. But the rest of the program could be reduced to this:

```
STORE_FREE_ACCT_NUMBERS;
FOR COUNT WHILE ~ERROR UNTIL ACCT NO <= 0
   REPEAT PROCESS_NEXT_ACCOUNT; END;
IF ERROR THEN TERMINATE_WITH_ERROR;
PRINT_STATISTICS;
```

There's a catch, of course -- these functions are not pre-defined, like PRODUCT or PRINT. We have to write the definitions ourselves, and submit them to the compiler as part of the program. Otherwise, the program will be rejected, with a message that reports that the function names are undefined.

Defining a function

Now we come to the crucial point. What does a function definition consist of? How is it written?

Fundamentally, a function definition is a sequence of CS-4 statements, which specify what is to be done each time the function is called. These statements are compiled along with the rest of the program;

they are executed when a function call is encountered during the program's execution.

The statements that make up a function definition have to be written in some particular order. When the function is called, the first executable statement in the definition is executed first; then control passes to subsequent statements in the definition, according to the rules for flow of control. After the last statement of the function definition is executed, control passes back to the place of the function call, and execution of the program continues from there.

A function definition can, therefore, be looked at as a sort of "sub-program". Control passes to this sub-program every time the function is called. When the sub-program is concluded, control passes back to the main program, which starts executing again where it left off.

We encountered the concept of a sub-program once before -- in introducing BEGIN blocks in Chapter 7. BEGIN blocks can also be thought of as sub-programs. However, the BEGIN block sub-program is executed when control passes to it by the usual rules for sequential flow of control. A function definition sub-program, by contrast, can have control passed to it only by a function call.

For the rest of this chapter, we'll concern ourselves with writing definitions for the particular functions we hypothesized for the billing program. In the process you will be able to get a more concrete appreciation of the working of function calls, executions, and returns.

Writing some FUNCTION definitions

We'll start by writing the FUNCTION definition for PRINT_STATISTICS, since its few essential features are common to all FUNCTIONs. It begins with a heading that resembles the start of a labelled BEGIN block:

```
PRINT_STATISTICS:
   FUNCTION;
```

Next comes the body of the definition -- the statements that are to be executed when PRINT_STATISTICS is called. These are:

```
PRINT; PRINT;
PRINT (COUNT || ' ACCOUNTS WERE PROCESSED');
PRINT (PAST_COUNT || ' ACCOUNTS HAD BALANCES PAST DUE');
```

Finally, the end of the FUNCTION is marked with an END statement, identical in form to the type that ends a labelled BEGIN:

```
END PRINT_STATISTICS;
```

Putting these together, we have the FUNCTION definition for PRINT_STATISTICS:

```
PRINT_STATISTICS;
   FUNCTION;
      PRINT; PRINT;
      PRINT (COUNT || ' ACCOUNTS WERE PROCESSED');
      PRINT (PAST_COUNT || ' ACCOUNTS HAD BALANCES PAST DUE');
   END PRINT_STATISTICS;
```

What happens now when the function call

```
PRINT_STATISTICS;
```

is executed? The call causes the flow of control to jump to the first executable statement in the body of the FUNCTION definition -- in this case, the statement that prints a blank line. Control then passes to subsequent statements within the FUNCTION definition, until it reaches the END. When END is reached, control returns to the point of the call, and the program continues from there. In our program, the statement that called PRINT_STATISTICS:

```
PRINT_STATISTICS;
```

is now finished. And, since that statement is the last one in the program, the whole program is finished. (However, if another statement followed the call to PRINT_STATISTICS, that statement would now be executed.)

Unlike BEGIN blocks, FUNCTION definitions are not executable statements. You cannot cause a function body to be executed by placing its FUNCTION and letting control pass to it in the normal sequence. A function definition is executed only when it is called from some other statement in the program. Since FUNCTION definitions are non-executable, they may appear in just those places where other non-executable statements -- such as declaration statements -- may appear. Customarily, they are all placed at the beginning or the end of a program. (Function calls, though, can appear in THEN clauses, in REPEAT statements, or anywhere other executable statements can appear.)

It sometimes helps to think of a function definition as a sort of declaration -- but one that declares a function name instead of a variable or constant name. We will go further into the declaration-like properties of FUNCTION definitions in Chapter 22. You will learn that both declaration statements and FUNCTION definitions have their meanings somewhat modified when they appear inside of BEGIN blocks or other FUNCTION definitions; so in the meantime you should play safe by keeping non-executable statements outside of BEGINs or FUNCTIONs.

Including the declaration statements and the PRINT_STATISTICS definition, the program now looks like this:

```
VARIABLES ACCT_NO, OLD_BAL, PAYMENT, PURCHASE, INTEREST,
   PAST_DUE, NEW_BAL, FREE_ACCT, FIRST, I, LAST ARE INTEGER,
   COUNT, PAST_COUNT ARE INTEGER ::= 0,
VARIABLE ERROR_MESSAGE IS STRING(40),
   NAME IS STRING(20);
VARIABLE ERROR IS BOOLEAN ::= FALSE;
   FREE_ACCT_TABLE IS ARRAY(999) BOOLEAN ::= FALSE;

STORE_FREE_ACCT_NUMBERS;
FOR COUNT WHILE ~ERROR UNTIL ACCT_NO <= 0
   REPEAT PROCESS_NEXT_ACCOUNT; END;
IF ERROR THEN TERMINATE_WITH_ERROR;
PRINT_STATISTICS;

PRINT_STATISTICS:
   FUNCTION;
      PRINT; PRINT;
      PRINT (COUNT || ' ACCOUNTS WERE PROCESSED');
      PRINT (PAST_COUNT || ' ACCOUNTS HAD BALANCES PAST DUE');
   END PRINT_STATISTICS;
```

Before defining the remaining three functions, we introduce here a few useful terms. Program-level statements are those not contained within any block or function; other statements are said to be at lower levels. Analogously, a program-level variable is one whose declaration is at program level. And a program-level function is one whose defining FUNCTION is not contained within any other block or FUNCTION. Pre-defined

functions are also treated as program-level functions. Our program contains five executable program-level statements (some of them compound), and all the functions and variables are program-level.

The RETURN statement in FUNCTION bodies

It is often advantageous to terminate a FUNCTION before control passes normally to END. Whereas BEGIN blocks and REPEAT statements force termination by executing EXIT, FUNCTIONs require a different statement: RETURN. When RETURN is executed within a FUNCTION body, it causes execution of the function to be terminated. It has the same effect as passing control in sequence to the END of the FUNCTION body. RETURN thus results in an immediate return to the calling point -- hence its name.

RETURN enables the work of a function to come to an end at any of several different points in its body. Take STORE_FREE_ACCT_NUMBERS, for instance. Its task is completed upon any of three conditions:

1) An error is found in the input value for N.

2) An error is found in some account number input.

3) All N inputs are processed correctly.

We write the FUNCTION definition so that cases (1) and (2) cause a RETURN to be executed, while in case (3) control passes in the normal sequence to END:

```
STORE_FREE_ACCT_NUMBERS:
   FUNCTION;
      READ_LINE (N);
      IF N < 0 | N > 999 THEN
         BEGIN;
            ERROR := TRUE;
            ERROR_MESSAGE := 'NUMBER OF FREE ACCOUNTS ' ||
               'NEGATIVE OR >999';
            RETURN;
         END;
```

```
      FOR I THRU N
         REPEAT
            READ_LINE (FREE_ACCT);
            IF FREE_ACCT < 1 | FREE_ACCT > 999 THEN
               BEGIN;
                  ERROR := TRUE;
                  ERROR_MESSAGE := 'FREE ACCOUNT NUMBER IS ' ||
                     '<1 OR >999';
                  RETURN;
               END;
            FREE_ACCT_TABLE(FREE_ACCT) := TRUE;
         END;
   END STORE_FREE_ACCT_NUMBERS;
```

Note that the EXIT FROM NEXT_NUMBER statement in the original program is here replaced by a RETURN. (Be sure to keep RETURN and EXIT straight: the former terminates FUNCTIONs, the latter terminates only blocks and REPEATs. When a RETURN is inside a BEGIN block which is in a FUNCTION body, it terminates the BEGIN along with the FUNCTION -- in fact, the second RETURN in our example terminates both a BEGIN and a REPEAT. But there is no way to use an EXIT to cause a return from a function call.)

Terminating the program from within a function body

The function TERMINATE_WITH_ERROR is supposed to print out an error message and then terminate the program. But we have not yet shown how it can do the terminating. It certainly isn't enough to execute RETURN within TERMINATE_WITH_ERROR's function body -- that would just cause a return to the calling point. What's needed is a way to terminate the program from anywhere inside it -- even from within a function body.

Our need is satisfied by the pre-defined function TERMINATE, which we have used frequently already. Wherever it is located, its effect is to end execution of the program immediately.

Now we can write the error-handling function:

```
TERMINATE_WITH_ERROR:
   FUNCTION;
      IF COUNT = 0 THEN PRINT ('EXECUTION TERMINATED BY ' ||
         'ERROR BEFORE PROCESSING: ', ERROR_MESSAGE);
      ELSE PRINT ('EXECUTION TERMINATED BY ' ||
         'ERROR DURING PROCESSING: ', ERROR_MESSAGE);
      TERMINATE;
   END TERMINATE_WITH_ERROR;
```

TERMINATE is best used in functions like this one -- functions that are called only on an error or other exceptional conditions. If TERMINATE is used in too many functions, it becomes hard to tell where execution ends in normal cases, and so some of the clarity that comes from splitting a program into functions is lost.

Calling a function from within a function

So far all our program-level functions have been executed by calls from within program-level statements. But calls to program-level functions may also come from within FUNCTION bodies. In other words, during the execution of one function, it may call yet a second function; when this second function ends, it returns control to the point from which it was called in the first function.

You have already seen some examples of this -- when we call the pre-defined functions PRINT and READ_LINE from within other functions. But we can also define our own functions to be called within others. Consider the problem of defining PROCESS_NEXT_ACCOUNT. It is by far the longest and most complicated function of the program. To keep it from getting too long, we can postulate another function -- call it PROCESS_INPUT_NAME -- to do the processing and checking of NAME. By calling PROCESS_INPUT_NAME from within PROCESS_NEXT_ACCOUNT, we can keep the latter's definition to a reasonable size:

```
PROCESS_NEXT_ACCOUNT:
   FUNCTION;
      READ_LINE (NAME, ACCT_NO, OLD_BAL, PAYMENT, PURCHASE);
      PROCESS_INPUT_NAME;
      IF ERROR THEN RETURN;
      IF ACCT_NO <= 0 THEN RETURN;
      IF ACCT_NO > 999 THEN
         BEGIN;
            ERROR := TRUE;
            ERROR_MESSAGE := 'ACCOUNT HAS NUMBER >999';
            RETURN;
         END;
      IF OLD_BAL = 0 & PAYMENT = 0 & PURCHASE = 0 THEN
         BEGIN;
            PRINT (ACCT_NO, NAME);
            RETURN;
         END;
      IF FREE_ACCT_TABLE(ACCT_NO) | PAYMENT >= OLD_BAL THEN
         BEGIN;
            PAST_DUE := 0;
            INTEREST := 0;
         END;
      ELSE
         BEGIN;
            PAST_DUE := OLD_BAL - PAYMENT;
            INTEREST := PAST_DUE * 0.015;
            PAST_COUNT := PAST_COUNT + 1;
         END;
      NEW_BAL := OLD_BAL - PAYMENT + INTEREST + PURCHASE;
      PRINT (ACCT_NO, NAME, OLD_BAL, PAYMENT,
         PAST_DUE, INTEREST, PURCHASE, NEW BAL);
   END PROCESS_NEXT_ACCOUNT;
```

Each time PROCESS_NEXT_ACCOUNT is called, its first action is to call READ_LINE, which reads in the values and returns control to PROCESS_NEXT_ACCOUNT. The next statement is a call to PROCESS_INPUT_NAME, which does all the work on NAME. Eventually, PROCESS_INPUT_NAME will also return control to the point of call, and then the rest of PROCESS_NEXT_ACCOUNT will be executed. Of course, now we have to define PROCESS_INPUT_NAME;

but that's a simple matter:

```
PROCESS_INPUT_NAME:
   FUNCTION;
      FOR FIRST THRU 20 UNTIL NAME(FIRST) ~= ' ' REPEAT; END;
      IF NAME(FIRST) = ' ' THEN RETURN;
      FOR LAST FROM 20 BY -1 UNTIL NAME(LAST) ~= ' ' REPEAT; END;
      FOR I FROM FIRST THRU LAST
         REPEAT
            IF (NAME(I) < 'A' | NAME(I) > 'Z') &
               NAME(I) ~= ' ' & NAME(I) ~= '''' THEN
               BEGIN;
                  ERROR := TRUE;
                  ERROR_MESSAGE := 'INVLAID CHARACTER WITHIN '
                     || 'A NAME';
                  RETURN;
               END;
            END;          # OF REPEAT FOR I #
         IF FIRST ~= 1 THEN
            NAME := NAME(LAST - FIRST + 1 AT FIRST);
      END PROCESS_INPUT_NAME;
```

PROCESS_INPUT_NAME is ended when control passes in sequence to END, or when one of the two RETURNS is executed. Whichever way it ends, PROCESS_INPUT_NAME always returns control to the point of call. Since it is called from within PROCESS_NEXT_ACCOUNT, it can only return control to PROCESS_NEXT_ACCOUNT. There is no way PROCESS_INPUT_NAME can return control to the program level -- because it is never called from the program level.

Putting it all together

All the functions we want are now defined. It remains only to put them all together, to produce a program suitable for submission to the compiler. We reproduce the entire program below, to give you a feel for its size and appearance:

```
VARIABLES ACCT_NO, OLD_BAL, PAYMENT, PURCHASE, INTEREST,
   PAST_DUE, NEW_BAL, FREE_ACCT, FIRST, I, LAST ARE INTEGER,
   COUNT, PAST_COUNT ARE INTEGER ::= 0;
VARIABLE ERROR_MESSAGE IS STRING(40),
   NAME IS STRING(20);
VARIABLE ERROR IS BOOLEAN ::= FALSE,
   FREE_ACCT_TABLE IS ARRAY(999) BOOLEAN ::= FALSE;

STORE_FREE_ACCT_NUMBERS;
FOR COUNT WHILE ~ERROR UNTIL ACCT_NO <= 0
   REPEAT PROCESS_NEXT_ACCOUNT; END;
IF ERROR THEN TERMINATE_WITH_ERROR;
PRINT_STATISTICS;

PRINT_STATISTICS:
   FUNCTION;
      PRINT; PRINT;
      PRINT (COUNT || ' ACCOUNTS WERE PROCESSED');
      PRINT (PAST_COUNT || ' ACCOUNTS HAD BALANCES PAST DUE');
   END PRINT_STATISTICS;

STORE_FREE_ACCT_NUMBERS:
   FUNCTION;
      READ_LINE (N);
      IF N < 0 | N > 999 THEN
         BEGIN;
            ERROR := TRUE;
            ERROR_MESSAGE := 'NUMBER OF FREE ACCOUNTS ' ||
               'NEGATIVE OR >999';
            RETURN;
         END;
      FOR I THRU N
         REPEAT
            READ_LINE (FREE_ACCT);
            IF FREE_ACCT < 1 | FREE_ACCT > 999 THEN
               BEGIN;
                  ERROR := TRUE;
```

```
                    ERROR_MESSAGE := 'FREE ACCOUNT NUMBER IS ' ||
                       '<1 OR >999';
                    RETURN;
                 END;
              FREE_ACCT_TABLE(FREE_ACCT) := TRUE;
           END;
   END STORE_FREE_ACCT_NUMBERS;

TERMINATE_WITH_ERROR:
   FUNCTION;
      IF COUNT = 0 THEN PRINT ('EXECUTION TERMINATED BY ' ||
         'ERROR BEFORE PROCESSSING: ', ERROR_MESSAGE);
      ELSE PRINT ('EXECUTION TERMINATED BY ' ||
         'ERROR DURING PROCESSING: ', ERROR_MESSAGE);
      TERMINATE;
   END TERMINATE_WITH_ERROR;

PROCESS_NEXT_ACCOUNT:
   FUNCTION;
      READ_LINE (NAME, ACCT_NO, OLD_BAL, PAYMENT, PURCHASE);
      PROCESS_INPUT_NAME;
      IF ERROR THEN RETURN;
      IF ACCT_NO <= 0 THEN RETURN;
      IF ACCT_NO > 999 THEN
         BEGIN;
            ERROR := TRUE;
            ERROR_MESSAGE := 'ACCOUNT HAS NUMBER >999';
            RETURN
         END;
      IF OLD_BAL = 0 & PAYMENT = 0 & PURCHASE = 0 THEN
         BEGIN;
            PRINT (ACCT_NO, NAME);
            RETURN;
         END;
      IF FREE_ACCT_TABLE(ACCT_NO) | PAYMENT >= OLD_BAL THEN
         BEGIN;
            PAST_DUE := 0;
            INTEREST := 0;
         END;
```

```
      ELSE
         BEGIN;
            PAST_DUE := OLD_BAL - PAYMENT;
            INTEREST := PAST_DUE * 0.015;
            PAST_COUNT := PAST_COUNT + 1;
         END;
      NEW_BAL := OLD BAL - PAYMENT + INTEREST + PURCHASE;
      PRINT (ACCT_NO, NAME, OLD_BAL, PAYMENT,
         PAST_DUE, INTEREST, PURCHASE, NEW_BAL);
   END PROCESS_NEXT_ACCOUNT;

PROCESS_INPUT_NAME:
   FUNCTION;
      FOR FIRST THRU 20 UNTIL NAME(FIRST) ~= ' ' REPEAT; END;
      IF NAME(FIRST) = ' ' THEN RETURN;
      FOR LAST FROM 20 BY -1 UNTIL NAME(LAST) ~= ' ' REPEAT; END;
      FOR I FROM FIRST THRU LAST
         REPEAT
            IF (NAME(I) < 'A' | NAME(I) > 'Z') &
               NAME(I) ~= ' ' & NAME(I) ~= '''' THEN
               BEGIN;
                  ERROR := TRUE;
                  ERROR_MESSAGE := 'INVALID CHARACTER WITHIN '
                     || 'A NAME';
                  RETURN;
               END;
            END:              # OF REPEAT FOR I #
         IF FIRST ~= 1 THEN
            NAME := NAME(LAST - FIRST + 1 AT FIRST);
      END PROCESS_INPUT_NAME;
```

It is obvious that our new version is even longer than the old one. But length is not the main concern; the important factors are ease of programming and debugging, and readability.

We have already demonstrated how programs structured in this way are somewhat easier to write, once you master the concept

of a function. First you write a short, general program-level program, writing just a function call for each major task. Then you program each task in detail by writing a FUNCTION definition that does the work. If a FUNCTION definition (such as FUNCTION_NEXT_ACCOUNT in our example) is itself fairly complex, you can implement one or more of its tasks as functions, too. This process can be repeated indefinitely, so that even in the most complicated program you do not have to write any single FUNCTION definition of more than about a page in length.

A program structured this way is also easier to read. First you look at the program level part, to get a general idea of what it does. Then you can inspect individual function bodies to see how various tasks are implemented. No individual function body is ever so long that you cannot keep track of what is going on. The flow of control is clearly indicated by calls and RETURNs, which are easier to follow than the EXITs and complicated IFs that would have to be used otherwise.

Finally, it is easier to debug or otherwise modify a program structured into FUNCTIONs. If you take care that the tasks performed by individual functions are independent of each other, then you can often modify a single function without changes to any of the rest of the program. For instance, if customer names were being read in incorrectly, or if the format for input names were changed, you could probably rewrite parts of PROCESS_INPUT_NAME without changes to any of the rest of the program. You could even change the error messages without altering TERMINATE_WITH_ERROR, so long as you do not change the way ERROR and ERROR_MESSAGE are set.

Structuring a program in this way also sometimes makes it easier to catch unanticipated errors which are detected at execution time -- such as wrongly formatted input, array subscripts out of bounds, and the like. The error message that results always indicates which function was being executed when the error occurred. In a large program, that may make the error easier to track down.

17.0
ARGUMENTS TO FUNCTIONS

In the last chapter we mentioned that common pre-defined functions like SUM and PRINT take arguments in their function calls:

```
SUM (WEIGHT(N AT 1) * DISTANCE(N AT 1));
PRINT (ACCT_NO, NAME);
```

The actions these functions perform depend in part on the values of the arguments they are called with.

Programmer-defined functions may also take and use arguments. This chapter is in introduction to how such FUNCTIONs are written, and how arguments are used in function calls.

Passing argument values

Functions with arguments are fairly easy to define and call, once you understand the principles behind them. In particular, you need to know why arguments are needed, and how they are used. To get at the answers we must discuss a concept that we brought up at the end of the last chapter: the independence of functions. One of the advantages of dividing a program into FUNCTIONs, we said, comes from giving each function a different, independent task to perform. Then corrections or changes can often be made to one FUNCTION without necessitating changes in other functions. As a result, there is less chance of introducing new mistakes in the program each time you make changes.

Notice that we have said only that changes can "often" be made independently in a FUNCTION. Actually, the simple functions we introduced in the last chapter cannot easily be kept independent. In large part, the problem is due to the way variable and constant names are declared: they are all declared at program level, and many are shared between functions.

We can demonstrate the problem with an example. Look again at the definition of TERMINATE_WITH_ERROR from the last chapter:

```
TERMINATE_WITH_ERROR:
   FUNCTION;
      IF COUNT = 0 THEN PRINT ('EXECUTION TERMINATED BY ' ||
         'ERROR BEFORE PROCESSING: ', ERROR_MESSAGE);
      ELSE PRINT ('EXECUTION TERMINATED BY ' ||
         'ERROR DURING PROCESSING: ', ERROR_MESSAGE);
      TERMINATE;
   END TERMINATE_WITH_ERROR;
```

This function only works properly if a STRING variable named ERROR_MESSAGE is declared at program level, and if other functions in the program properly assign to ERROR_MESSAGE the proper error-message strings. Conceivably, we could decide to make some changes involving ERROR_MESSAGE, such as

1) declaring a variable with a different name -- ERROR_STRING, maybe -- in its place; or

2) declaring it as an ARRAY(4) STRING(40), each element of which is initialized to a different error message; then printing out the appropriate element of the array when an error is found.

Either of these changes would require corrections to TERMINATE_WITH_ERROR to keep it working correctly.

We can reason similarly with regard to the variable COUNT. If the program logic were changed so that COUNT would be -1 before processing began, and 0 or greater during processing, then the old TERMINATE_WITH_ERROR would no longer work correctly.

We can restate the trouble we are running up against in a more revealing way. In order to operate, TERMINATE_WITH_ERROR needs two essential

pieces of information: whether or not the error occurred before account processing began, and what the error-message string is. The function gets this information by sharing the variables COUNT and ERROR_MESSAGE with other functions and with the program-level program. If changes are made to the shared variables elsewhere in the program, then changes will probably have to be made to TERMINATE_WITH_ERROR, too.

Functions with arguments are a solution to this problem. Arguments are an alternative way of passing data to a function -- a way that does not require name-sharing. When you make a function call like

```
SUM (WEIGHT(N AT 1) * DISTANCE(N AT 1))
```

the body of SUM does not have to use the names WEIGHT, DISTANCE, and N. Instead, the value of the argument:

```
WEIGHT(N AT 1) * DISTANCE(N AT 1)
```

is assigned to a special kind of variable called a parameter. The parameter is a variable that belongs to SUM; it cannot be referred to anywhere in the program outside of SUM's FUNCTION body. In its body, SUM adds together all the elements of the parameter variable. This computation is entirely unaffected by any changes that are made in the declarations of WEIGHT, DISTANCE or N -- and in fact, we made quite a few such changes in writing the center-of-gravity program. No matter how the argument to SUM is changed, its value is always assigned to the parameter variable; and since the parameter is not affected by changes to the rest of the program, it can always be manipulated by SUM in exactly the same way.

This is the theory behind function arguments. As with other theories, it becomes clearer in practice; so we will move on now to some examples.

A one-argument function

Let us rewrite the function TERMINATE_WITH_ERROR, so that it gets the error-message value by argument passage instead of by referring to the name ERROR_MESSAGE.

The first step is to declare a parameter of mode STRING(40). The declaration is placed in parentheses within the FUNCTION heading:

```
TERMINATE_WITH_ERROR:
   FUNCTION (MESSAGE_OUTPUT IS STRING(40));
```

Here MESSAGE_OUTPUT is defined as the parameter for TERMINATE_WITH_ERROR. When the function is called, the value of its argument is assigned to MESSAGE_OUTPUT. In the present case, the argument is always a STRING(40) containing an error message; so MESSAGE_OUTPUT is also declared as a STRING(40).

In the body of the function, there is just one change. It is now the parameter, MESSAGE_OUTPUT, that gets printed out, instead of the program-level variable ERROR_MESSAGE. The new FUNCTION definition in its entirety is therefore as follows:

```
TERMINATE_WITH_ERROR:
   FUNCTION (MESSAGE_OUTPUT IS STRING(40));
      IF COUNT = 0 THEN PRINT ('EXECUTION TERMINATED BY ' ||
         'ERROR BEFORE PROCESSING: ', ERROR_MESSAGE);
      ELSE PRINT ('EXECUTION TERMINATED BY ' ||
         'ERROR DURING PROCESSING: ', ERROR_MESSAGE);
      TERMINATE;
   END TERMINATE_WITH_ERROR;
```

How is this new function called? Instead of the program-level statement

```
IF ERROR THEN TERMINATE_WITH_ERROR;
```

which calls the function with no arguments, it is now necessary to write

```
IF ERROR THEN TERMINATE_WITH_ERROR (ERROR_MESSAGE);
```

Now, if ERROR is TRUE, the function call

```
TERMINATE_WITH_ERROR (ERROR_MESSAGE)
```

is executed. The first action performed by the call is argument passage:

the value of the argument (ERROR_MESSAGE) is assigned to the parameter variable (MESSAGE_OUTPUT). After this assignment is made, the function body is executed, and MESSAGE_OUTPUT gets printed out by one or the other PRINT statement. Since MESSAGE_OUTPUT has been assigned the value of the argument, ERROR_MESSAGE, it is actually the value of ERROR_MESSAGE that ends up being printed out.

Since MESSAGE_OUTPUT is a parameter, it may not be used in the billing program outside of the FUNCTION definition for TERMINATE_WITH_ERROR. So changes in the rest of the program cannot directly affect the parameter, or the way it is used in the function body.

What about the changes to ERROR_MESSAGE that we hypothesized a few sections back? They can be taken care of by just changing the argument to the function call. If ERROR_STRING is used instead of ERROR_MESSAGE, then the program-level statement is just changed to read:

```
IF ERROR THEN TERMINATE_WITH_ERROR (ERROR_STRING);
```

If ERROR_MESSAGE is made an ARRAY(4) STRING(40), and ERROR_NO is a value from 1 to 4 that tells which element to print out, then the proper statement is:

```
IF ERROR THEN TERMINATE_WITH ERROR (ERROR_MESSAGE(ERROR_NO));
```

If ERROR_MESSAGE is changed to a STRING(35), one can use:

```
IF ERROR THEN TERMINATE_WITH_ERROR (ERROR_MESSAGE || '      ');
```

In each case, the argument value is a STRING(40), and is assigned to the parameter; there is no need to change the declaration of the parameter, or to change the way it is used in the function body.

One other alternative is worth mentioning. It is possible to eliminate the BOOLEAN variable ERROR from the billing program entirely, so that instead of setting

```
ERROR := TRUE;
```

when an error occurs, one can execute TERMINATE_WITH_ERROR directly.

Here is how such a change would be made to STORE_FREE_ACCT_NUMBERS:

```
STORE_FREE_ACCT_NUMBERS:
   FUNCTION;
      READ_LINE (N);
      IF N < 0 | N > 999 THEN
         TERMINATE_WITH_ERROR ('NUMBER OF FREE ACCOUNTS ' ||
            'NEGATIVE OR >999');
      FOR I THRU N
         REPEAT
            READ_LINE (FREE_ACCT);
            IF FREE_ACCT < 1 | FREE_ACCT > 999 THEN
               TERMINATE_WITH_ERROR ('FREE ACCOUNT NUMBER IS ' ||
                  '<1 OR >999       ');
            FREE_ACCT_TABLE(FREE_ACCT) := TRUE;
         END;
   END STORE_FREE_ACCT_NUMBERS;
```

Similar changes would have to be made to handle errors in PROCESS_NEXT_ACCOUNT and PROCESS_INPUT_NAME. Note that, when TERMINATE_WITH_ERROR is called directly, the STRING-literals for the messages can themselves be used as arguments; the variable ERROR_MESSAGE is not needed at all. MESSAGE_OUTPUT still serves as the function's parameter, of course, and is unaffected by these changes.

Functions with more than one argument

Functions are not limited to one parameter (and thus one argument). A FUNCTION heading may contain a list of any number of parameter declarations, separated by commas. Thus several arguments of different modes may be passed to a function when it is called.

As an example, we can eliminate the use of the shared variable COUNT in TERMINATE_WITH_ERROR, by declaring a second parameter of mode BOOLEAN:

```
TERMINATE_WITH_ERROR:
   FUNCTION (MESSAGE_OUTPUT IS STRING(40),
      PROCESSING IS BOOLEAN);
      IF ~PROCESSING THEN
         PRINT ('EXECUTION TERMINATED BY ' ||
            'ERROR BEFORE PROCESSING: ', MESSAGE_OUTPUT);
      ELSE
         PRINT ('EXECUTION TERMINATED BY ' ||
            'ERROR DURING PROCESSING: ', MESSAGE_OUTPUT);
      TERMINATE;
   END TERMINATE_WITH_ERROR;
```

If FALSE is passed to the parameter PROCESSING, the function assumes the error occurred before processing; if TRUE is passed, it assumed the error was during processing.

In the structured billing program as it was written in the last chapter, this function would be called by the program-level statement:

```
IF ERROR THEN
   TERMINATE_WITH_ERROR (ERROR_MESSAGE, COUNT > 0);
```

If the logic of the program were modified, so that COUNT was negative before processing, and non-negative during, this statement would just have to be changed to

```
IF ERROR THEN
   TERMINATE_WITH_ERROR (ERROR_MESSAGE, COUNT >= 0);
```

Finally, if we wanted to call the function directly on finding an error -- as we proposed in the last section -- the BOOLEAN argument could be a literal. For instance, this would be the first error test in STORE_FREE_ACCT_NUMBERS:

```
IF N < 0 | N > 999 THEN
   TERMINATE_WITH_ERROR ('NUMBER OF FREE ACCOUNTS ' ||
      'NEGATIVE OR 999', FALSE);
```

The second argument here is FALSE, because in STORE_FREE_ACCT_NUMBERS processing has not yet begun.

Rules for argument passage

Any number of parameters, we have said, may be declared in a function heading, and they may be declared of any mode. Several parameters of the same mode may be declared together with ARE:

```
ABC:
   FUNCTION (A, B, C ARE INTEGER);
```

This is equivalent to declaring the same three in the same order separately:

```
ABC:
   FUNCTION (A IS INTEGER, B IS INTEGER, C IS INTEGER);
```

When a function call with arguments is executed, the first value in the argument list is assigned to the first parameter in the declaration list, the second argument is passed to the second parameter, and so on. The number of argument values in the call must equal the number of parameters in the function definition. If a call has the wrong number of arguments, the compiler rejects it, and the program cannot be run.

An argument value must have the same mode as the parameter it is assigned to, unless the parameter specification allows more tolerance. (That subject is not covered in this volume of the Primer.) If a function has arrayed parameters of a given size, such as

```
CENTER:
   FUNCTION (WEIGHT, DISTANCE ARE ARRAY(5) REAL);
```

then the arguments must have the same arrayness as the parameters. The arguments to CENTER must be ARRAY(5) REAL; they may not be ARRAY(10) REAL, or ARRAY(5,3) REAL, or just REAL. The same restrictions apply for STRING-mode parameters, In the TERMINATE_WITH_ERROR example:

```
TERMINATE_WITH_ERROR:
   FUNCTION (MESSAGE_OUTPUT IS STRING(40),
      PROCESSING IS BOOLEAN);
```

the first argument must be a STRING(40); it may not be a STRING(35) or STRING(45).

Parameters may generally be used in a function body in all ways other variables may. However, there is an important restriction on the type of parameters we are using in this chapter: they may not be assigned values within the function body. In other words, our parameters may not appear on the left-hand side of an assignment operator, or as arguments to READ_LINE. The same restrictions apply to subscripted array and STRING parameters. Ways do exist to declare parameters so that these restriction are lifted; the next chapter will demonstrate how.

Array parameters changeable size

Some properties of parameters differ from those of program-level variables. The most important difference is the one you already know: parameters can be used only in the function where they are defined. Another difference concerns array parameters. Program-level arrays must be declared of some fixed size and rank; thus they have the same arrayness throughout a program. Parameter arrays may be declared with a variable size in one or more dimensions. As a result the size of a parameter array may vary from call to call.

Variable-sized parameters prove useful in numerous applications. As an example, we can re-structure the center-of-gravity program of Chapters 10 and 11, using FUNCTIONs. This program accepts input data for N weights, where N can range from 1 to 25. The program-level part of the program could look like this:

```
VARIABLES WT, DIS ARE ARRAY(25) REAL,
   CENTER IS REAL,
   N IS INTEGER;
REPEAT
   READ_LINE (N);
   IF N = 0 THEN TERMINATE;
   IF N < 0 | N > 25 THEN
      BEGIN;
         PRINT ('*****ERROR: NUMBER of WEIGHTS IS ' ||
            'NEGATIVE OR >25');
         TERMINATE;
      END;
   READ_LINE (WT(N AT 1), DIS(N AT 1));
   FIND_CENTER_OF_GRAVITY (WT(N AT 1), DIS(N AT 1));
   PRINT (WT(N AT 1), DIS(N AT 1), CENTER);
END;
```

We have left only one function, FIND_CENTER_OF_GRAVITY, to be defined; it needs two array parameters. But notice that the arguments to FIND_CENTER_OF_GRAVITY may range from ARRAY(1) REAL to ARRAY(25) REAL, depending on the last input value of N. So if the function were defined with parameters of fixed size, the program would not work: arguments of any other size would be errors.

The solution is to declare the parameters as ARRAY(N) REAL, so that FIND_CENTER_OF_GRAVITY is defined as follows:

```
FIND_CENTER_OF_GRAVITY:
   FUNCTION (WEIGHT, DISTANCE ARE ARRAY(N) REAL);
      IF SUM (WEIGHT) = 0 THEN
         BEGIN;
            PRINT ('*****ERROR: WEIGHT SUM = 0');
            TERMINATE;
         END;
      CENTER := SUM (WEIGHT * DISTANCE) / SUM (WEIGHT);
   END FIND_CENTER_OF_GRAVITY;
```

Each time this function is called, WEIGHT and DISTANCE are made arrays of size N. If N varies from call to call, so does the size of the parameters. However, during execution of FIND_CENTER_OF_GRAVITY, the size of WEIGHT and DISTANCE is fixed. It cannot be changed, even if the function changes N.

There is a way to restate the rules for program-level and parameter array sizes, so that they are fairly analogous. Program-level arrays have their size fixed, or resolved, when the program's execution begins; their size cannot be changed thereafter. Parameter arrays have their size resolved each time the function that defines them is called; their size cannot be changed while the function is being executed. (The rank of any array variable, by the way, is fixed through the program.)

The concept of varying-size parameters can of course be extended to arrays of size 2 or greater. Here is an example: a version of the 3-dimensional center-of-gravity program of Chapter 13. Restructured using a function with varying-size parameters, the program can take from 1 to 25 weights:

```
VARIABLE WT IS ARRAY(25) REAL,
   DIS IS ARRAY(25,3) REAL,
   WEIGHT_SUM, PRODUCT_SUM, CENTER ARE REAL,
   I, N ARE INTEGER;
REPEAT
   READ_LINE (N);
   IF N = 0 THEN TERMINATE;
   IF N < 0 | N > 25 THEN
      BEGIN;
         PRINT ('*****ERROR: NUMBER OF WEIGHTS IS ' ||
            'NEGATIVE OR 25');
         TERMINATE;
      END;
   READ_LINE (WT(N AT 1), DIS(N AT 1));
   FIND_CENTER_OF_GRAVITY (WT(N AT 1), DIS(N AT 1));
   PRINT (WT(N AT 1), DIS(N AT 1), CENTER);
END;
```

```
FIND_CENTER_OF_GRAVITY:
   FUNCTION (WEIGHT IS ARRAY(N) REAL,
      DISTANCE IS ARRAY(N,3) REAL);
      WEIGHT_SUM := 0;
      PRODUCT_SUM := 0;
      FOR I THRU N
         REPEAT
            WEIGHT_SUM := WEIGHT_SUM + WEIGHT(I);
            PRODUCT_SUM :=
               PRODUCT_SUM + WEIGHT(I) * DISTANCE(I);
         END;
      IF WEIGHT_SUM = 0 THEN
         BEGIN;
            PRINT ('*****ERROR: WEIGHT SUM = 0');
            TERMINATE;
         END;
      CENTER := PRODUCT_SUM / WEIGHT_SUM;
   END FIND_CENTER_OF_GRAVITY;
```

Another application would be in our temperature-grid program, if the size of the grid could vary in one or both dimensions.

Automatic resolution of parameter size

There is an unfortunate drawback to the programs of the previous section. The program-level variable N is used to resolve the sizes of the array paramters. N is thus a shared variable -- exactly what parameters are supposed to eliminate. Worse yet, there is no way that N can be passed as an argument of a function such as FIND_CENTER_OF_GRAVITY.

Let's see why. Suppose we try to make N one of the arguments to the function:

```
FIND_CENTER_OF_GRAVITY (WT(N AT 1), DIS(N AT 1), N);
```

Then we have to add an INTEGER parameter, call it SIZE, to the function heading:

```
FIND_CENTER_OF_GRAVITY:
   FUNCTION (WEIGHT IS ARRAY(SIZE) REAL,
      DISTANCE IS ARRAY(SIZE) REAL, SIZE IS INTEGER);
```

In order for this to work, a call to FIND_CENTER_OF_GRAVITY would have to proceed in the following manner: (1) the value of N is passed to SIZE; (2) the value of SIZE is used to resolve the sizes of WEIGHT and DISTANCE; (3) values are passed to WEIGHT and DISTANCE. But the compiler does not work this way. It does all the necessary size resolution before any argument passage. Thus SIZE has not yet been assigned a value when the sizes of WEIGHT and DISTANCE are to be resolved, and the above heading is in error.

One way to circumvent these problems is to use automatic size resolution. This feature permits you to say that the size of an array parameter is to be set automatically to the size of the argument that is passed. Automatic resolution is indicated by writing a * instead of a dimension size in the array parameter declaration:

```
FIND_CENTER_OF_GRAVITY:
   FUNCTION (WEIGHT, DISTANCE ARE ARRAY(*) REAL);
```

This heading says that WEIGHT and DISTANCE are 1-dimensional arrays, and that when the function is called they will take their sizes from those of the corresponding arguments. If we add the same body as before, FIND_CENTER_OF_GRAVITY will do exactly the same calculations, but with no reference to the name N.

Multi-dimensional array parameters may have one or more dimension sizes resolved automatically. For instance, in the three-dimensional center-of-gravity program, FIND_CENTER_OF_GRAVITY can be defined with:

```
FIND_CENTER_OF_GRAVITY:
   FUNCTION (WEIGHT IS ARRAY(*) REAL,
      DISTANCE IS ARRAY(*,3) REAL);
```

This says that DISTANCE will take its first-dimension size from the corresponding argument; but the argument's second-dimension size must be 3. On the other hand,

```
FIND_CENTER_OF_GRAVITY:
   FUNCTION (WEIGHT IS ARRAY(*) REAL,
      DISTANCE IS ARRAY(*,*) REAL);
```

says that both of DISTANCE's dimension sizes will be taken from the argument. You might want to try using this heading in a program that handles a variable number of weights in a space of a variable number of dimensions.

The pre-defined function SIZE

It is sometimes necessary to refer to the size of an array parameter within the function body. This was done in the 3-dimensional FIND_CENTER_OF_GRAVITY, in the statement:

```
FOR I THRU N REPEAT ... END;
```

where N was the first-dimension size of WEIGHT and DISTANCE. It would be better not to use N, however, since it is a program-level variable.

One alternative is to use the pre-defined function SIZE. It takes one or two arguments. The first (or only) argument is an array of any element-mode; SIZE calculates the size of one of its dimensions. The second argument is an INTEGER that tells which dimension's size is returned. If there is no second argument, SIZE returns the size of the first dimension. Thus in FIND_CENTER_OF_GRAVITY we could write

```
FOR I THRU SIZE (DISTANCE, 1) ...
```

or

```
FOR I THRU SIZE (DISTANCE) ...
```

instead of using N.

SIZE is also valuable for checking parameters which are size-resolved automatically. For instance, FIND_CENTER_OF_GRAVITY requires that the number of weight values be equal to the number of distance values. But the heading:

```
FIND_CENTER_OF_GRAVITY:
   FUNCTION (WEIGHT IS ARRAY(*) REAL,
      DISTANCE IS ARRAY(*,3) REAL);
```

imposes no such restriction. It accepts any first-dimension sizes for WEIGHT and DISTANCE. So we ought to write an extra test in the function body:

```
IF SIZE (WEIGHT) ~= SIZE (DISTANCE) THEN
   BEGIN;
      PRINT (*****ERROR: 'NUMBER OF WEIGHTS DOES NOT ' ||
         'EQUAL NUMBER OF DISTANCES');
      TERMINATE;
   END;
```

which catches the error when the parameter sizes do not match. Of course, this can only happen when the argument sizes are incorrect -- when there is some sort of error in writing the function call.

STRING parameters of unresolved length

Since STRING values are basically arrays of characters, STRING parameters of unresolved length may be produced by the same means used for arrays of unresolved size. An expression containing a variable may be used instead of an INTEGER-literal to specify the string's length. Or a * may be written instead of a length, to indicate automatic length resolution.

One application of this would be to write the function TERMINATE_WITH_ERROR as:

```
TERMINATE_WITH_ERROR:
   FUNCTION (MESSAGE_OUTPUT IS STRING(*));
```

This eliminates the need to make every error message come out exactly 40 characters long, and so does away with bugs that result from careless slips in counting.

A pre-defined function LENGTH performs the same service for strings that SIZE does for arrays declared with ARRAY. LENGTH takes a single STRING-mode argument, and returns the length of the argument.

18.0
PARAMETER BINDINGS

All the parameters we showed how to declare in the last chapter have the same basic properties. They are passed an argument value each time their function is called, and, because no new value can be assigned to them, they keep that value throughout the execution of the function.

In many applications it is desirable to declare parameters with different or additional properties. Sometimes one wishes not to send a value to a parameter, but to have that parameter pass a value back to the argument when the function returns. Sometimes one wishes to pass a value to a parameter, then have the function modify that value, and finally have the new value returned. To meet these various requirements, CS-4 provides a means of declaring different types of parameters (or, more exactly, different types of parameter "binding" -- ways of binding parameters to their arguments). This chapter shows how parameter bindings are indicated in a declaration, and explains the meanings of three types in addition to the one you already know.

INPUT parameter binding

Each type of parameter in CS-4 has a name. The parameters we declared in the preceding chapter are called INPUT parameters. We have not had to mention this fact before now, because when no type of binding is specified in a parameter declaration the compiler assumes an INPUT binding is intended. (INPUT is thus said to be the default parameter binding.)

How is a parameter binding indicated explicitly? By placing the parameter-binding name after the mode name in the parameter's declaration. For example:

```
TERMINATE_WITH_ERROR:
   FUNCTION (MESSAGE_OUTPUT IS STRING(40) INPUT,
      PROCESSING IS BOOLEAN INPUT);
```

Here MESSAGE_OUTPUT and PROCESSING are explicitly declared to be INPUT parameters. Including the name INPUT here is optional, of course -- because INPUT is the default binding. To use other types, however, you must name them in the declaration.

INOUT parameter binding

We have seen that INPUT binding allows data to pass into a function. Sometimes, however, we need a parameter which also permits the function to assign new values to the parameter variables and which will pass the new values back to the argument variables when the function returns. The INOUT binding is for use in this situation. Consider, for example, the PROCESS_INPUT_NAME function from the billing program. This function moves the leading blank characters in NAME to the other end of the string. If the function is re-written so that the value of NAME is assigned to an INOUT parameter, the function can have the added feature of being able to accept strings of any length. The original value of NAME will be passed to the function's parameter variable, the function will make the modification to that variable, and the new value will be passed back to NAME when the function returns. The PROCESS_INPUT_NAME function, using INOUT parameter binding, may be written like this:

```
PROCESS_INPUT_NAME:
   FUNCTION (STR IS STRING(*) INOUT);
      FOR FIRST THRU SIZE (STR) UNTIL STR(FIRST) ~= ' ';
         REPEAT; END;
      IF STR(FIRST) = ' ' THEN RETURN;
      FOR LAST FROM SIZE (STR) UNTIL STR(FIRST) ~= ' ';
         REPEAT; END;
            . . .
      STR := STR(LAST - FIRST + 1 AT FIRST);
   END PROCESS_INPUT_NAME;
```

The function may be invoked by the call

```
PROCESS_INPUT_NAME (NAME);
```

and the value of NAME after control returns from PROCESS_INPUT_NAME will be the new value of STR. The INPUT binding could not have been

used here, because INPUT does not allow values of parameters to be changed within the function.

OUTPUT parameter binding

OUTPUT parameter binding, as you might guess from the name, is used in situations in which the function makes no use of the value passed to the parameter in the call, but does assign to it a new value to be passed back when the function returns.

One application of OUTPUT binding could be in a function named GET_NEW_ACCT_NO, to be invoked when the account being processed is a new one that does not already have an account number assigned to it. ACCT_NO could be the argument of the call to the function:

```
GET_NEW_ACCT_NO (ACCT_NO);
```

and the purpose of the function will be to pass back to ACCT_NO a number which is not already assigned to somcone else's account.

The details of how the function body comes up with the new account number need not concern us here. Perhaps the program could keep a list of obsolete or unused account numbers, and the function could remove one of the numbers from the list and assign it to the new customer's ACCT_NO. Or, perhaps, the function could designate for the new customer an ACCT_NO one higher than the highest account number already in use. In any case, the heading and end statement of GET_NEW_ACCT_NO could look like this:

```
GET_NEW_ACCT_NO:
   FUNCTION (NEW_ACCT_NO IS INTEGER OUTPUT);
      . . .
      . . .
   END GET_NEW_ACCT_NO;
```

One way to indicate that a customer needs a new account number is to place in the input data a "dummy" account number of zero along with every new customer's name. That value of zero can be used to signal that the GET_NEW_ACCT_NO function is to be called:

```
IF ACCT_NO = 0 THEN GET_NEW_ACCT_NO (ACCT_NO);
```

When GET_NEW_ACCT_NO is called, the value of ACCT_NO that is passed to the NEW_ACCT_NO parameter is lost. GET_NEW_ACCT_NO assigns an appropriate integer value to NEW_ACCT_NO. When the function returns, that value is passed back to ACCT_NO, and is the new customer's account number. We are able to use OUTPUT as the parameter binding here, because the function makes no use of the old value of ACCT_NO.

A restriction on arguments sent to INOUT and OUTPUT parameters

INOUT and OUTPUT parameters are subject to an important restriction which does not apply to INPUT ones. An INOUT or OUTPUT parameter may not be passed an argument which is a literal, a CONSTANT name, or an expression using any of the operators we have defined so far. None of these objects can legally be assigned a value, so there is no way to pass a value back to them.

Variables (including subscripted variables) can be used as arguments to INOUT and OUTPUT parameters. (This is nothing but a generalization of the rules we gave for READ_LINE, a function that happens to pass values back to all its arguments.)

In other respects, INOUT and OUTPUT parameters come under the same rules as INPUT ones. They may not be referred to outside of the function where they are declared. If they are arrays or strings, they may have unresolved size (length), or may be subject to automatic size (length) resolution.

If a function has more than one parameter, it is permissible to give each a different binding. For instance, in

```
STORE_FREE_ACCT_NUMBERS:
   FUNCTION (NUMBER_OF_INPUTS IS INTEGER,
      TABLE IS ARRAY(*) BOOLEAN OUTPUT);
```

NUMBER_OF_INPUTS is an INPUT parameter (by default), while TABLE is an OUTPUT parameter whose size is automatically resolved. A function with this heading might be called from the billing program with:

```
READ_LINE (N);
STORE_FREE_ACCT_NUMBERS (N, FREE_ACCT_TABLE);
```

COPYIN parameters

Occasionally it is useful to be able to modify the value of a parameter that is only used for passing data into a function. Such a parameter may be declared with a parameter type named COPYIN.

As its name implies, the argument value passed to a COPYIN parameter is "copied" into a separate storage space. Subsequent changes (such as assignments) to the parameter change just this copy; they do not change the argument's value. INPUT parameters, by contrast, are not always passed a copy. Instead, in some cases, they share storage space with their arguments. Because it is impossible for a programmer to tell when an INPUT parameter is passed a copy and when it is not, assignments to INPUT parameters are forbidden entirely.

In other respects, the properties of COPYIN and INPUT parameters are identical. In particular, the argument to a COPYIN parameter may be any sort of expression, not just a variable.

Advanced topics

In addition to the four parameter bindings explained in this chapter, CS-4 has five others, which give the programmer explicit control over whether the function "copies" the argument values into separate storage. How to use those bindings is explained in the Language Reference Manual.

Some of the pre-defined functions can do more with their arguments than any of the functions we have shown how to define. PRINT, for instance, can take a varying number of arguments. It is possible to define your own functions that do this, as well as perform different actions depending on whether certain arguments are of certain modes. But to define those kinds of functions, you need to use forms other than the ones we have been describing. These forms are described in the Primer's second volume, and in the Language Reference Manual.

19.0

RETURN VALUES OF FUNCTIONS

A function has several means of making its internal data -- values it has calculated, or values it has read in -- available to the rest of the program. Two of these you have already learned to use. The values can be assigned to program-level variables, or they can be assigned to INOUT or OUTPUT parameters.

There is a third method, which you have only seen so far in predefined functions. A function may have a return value -- a value which is assigned to the function call itself. The call of such a function can be used in expressions to represent the return value, just like a variable, constant, or literal is used to represent a value:

```
IF ANY (DISTANCE < -5) | ANY (DISTANCE > 5) THEN RETURN;
   . . .
FOR I THRU SIZE (DISTANCE, 1) REPEAT ... END;
   . . .
PRODUCT_SUM := SUM (WEIGHT(N AT 1) * DISTANCE(N AT 1));
```

This chapter explains how functions are defined to have return values, and how they determine what value is to be returned.

Why return values?

Any piece of data that can be a return value of a function can also be passed out by assigning it to an INOUT or OUTPUT parameter. Thus the concept of return values doesn't add much to the power of CS-4 functions. What it does improve is the clarity of certain function calls, and hence the ease with which a program can be written and read.

What sort of function lends itself to using a return value? Usually it is one whose task is to take one or more arguments, and to compute a single result. SUM, for instance, takes an array and computes the sum of its elements. SIZE takes an array, and an INTEGER; it interprets the latter to refer to a particular dimension of the former, and produces another INTEGER equal to the size of that dimension.

If you're familiar with algebra, you can look at the matter from another point of view. A "function" in algebra is a method of taking any value or combination of values within a certain domain, and producing a unique result value. The notation for such "functions":

```
f (a)
sin (x + y)
g (x,y,z)
```

is a lot like the CS-4 notation. Each expression like f (a) represents a calculated value, which may be subject to operations like addition and multiplication:

$$f\ (a)\ \cdot\ x_1\ +\ \sin\ (x_2)$$

So these "functions" are indeed used very much like CS-4 functions that have return values. We can turn the analogy around and say that CS-4 functions most apt to have return values are ones which calculate a value which is some unique "function" of their arguments. (But whereas algebraic "functions" are defined by English and traditional mathematical notation, CS-4 functions are defined by statements of CS-4.)

Of the functions we have been using as examples, there are a few that could be usefully defined to have a return value. FIND_CENTER_OF_GRAVITY, for instance, calculates the center of gravity's position from the values of two array arguments (weights and distances). PROCESS_INPUT_NAME, from the billing program, produces a unique reformatted string for each string passed to it in the variable NAME.

On the other hand, any function that returns more than one value is better written with OUTPUT or INOUT parameters.

There are exceptions to these guidelines, of course, but they're a good start in deciding where to make use of what you learn in this chapter.

Defining a function to have a return value

Our first example uses the center-of-gravity problem. Previously we defined a function FIND_CENTER_OF_GRAVITY, which computed the position of the center from the parameters WEIGHT and DISTANCE. The computed value was passed back to program level by assigning it to a program-level variable, CENTER. The entire program, including the function definition, looked like this:

```
VARIABLES WT, DIS ARE ARRAY(25) REAL,
   CENTER IS REAL,
   N IS INTEGER;
REPEAT
   READ_LINE (N);
   IF N = 0 THEN TERMINATE;
   IF N < 0 | N > 25 THEN
      BEGIN;
         PRINT ('*****ERROR:  NUMBER OF WEIGHTS IS ' ||
            'NEGATIVE OR > 25');
         TERMINATE;
      END;
   READ_LINE (WT(N AT 1), DIS(N AT 1));
   FIND_CENTER_OF_GRAVITY (WT(N AT 1), DIS(N AT 1));
   PRINT (WT(N AT 1), DIS(N AT 1), CENTER);
END;

FIND_CENTER_OF_GRAVITY:
   FUNCTION (WEIGHT, DISTANCE ARE ARRAY(*) REAL);
      IF SUM (WEIGHT) = 0 THEN
         BEGIN;
            PRINT ('*****ERROR:  WEIGHT SUM = 0');
            TERMINATE;
         END;
      CENTER := SUM (WEIGHT * DISTANCE) / SUM (WEIGHT);
   END FIND_CENTER_OF_GRAVITY;
```

We want to replace the function in this program with a revised one, call it GRAVITY_CENTER, which passes back the position of the center as a return value. Then we can replace the call to FIND_CENTER_OF_GRAVITY at program level with

```
CENTER := GRAVITY_CENTER (WT(N AT 1), DIS(N AT 1));
```

Alternatively, we could eliminate CENTER entirely, by placing the call in the PRINT statement:

```
PRINT (WT(N AT 1), DIS(N AT 1),
   GRAVITY_CENTER (WT(N AT 1), DIS(N AT 1)));
```

(Notice that here the call to GRAVITY_CENTER is itself an argument to the function PRINT; the value returned by the former function is passed directly to the latter.)

Specifying a return mode

Every function which returns a value must specify a return mode in its heading. This is done by placing a mode name after the parameter declarations. For instance:

```
GRAVITY_CENTER:
   FUNCTION (WEIGHT, DISTANCE ARE ARRAY(*) REAL) REAL;
```

Here we indicate that GRAVITY_CENTER returns a REAL value.

Any mode may be returned by a function. Thus the return mode may be an array:

```
GRAVITY_CENTER_3:
   FUNCTION (WEIGHT IS ARRAY(*) REAL,
      DISTANCE IS ARRAY(*,3) REAL) ARRAY(3) REAL;
```

or a string:

```
PROCESS_INPUT_NAME:
   FUNCTION (INPUT IS STRING(40)) STRING(40);
```

If a function has no parameters, the return mode comes directly after the word FUNCTION:

```
RANDOM:
   FUNCTION REAL;
```

Indicating the return value

A function that returns a value has to indicate explicitly what value is returned. This is done by using a form of the RETURN statement. The returned value is indicated by an expression which follows the word RETURN:

```
RETURN SUM (WEIGHT * DISTANCE) / SUM (WEIGHT);
```

This is the RETURN statement for the function GRAVITY_CENTER. Like the simple RETURNs we used before, it causes control to be passed back to the point of call. But before it does so, the expression

```
SUM (WEIGHT * DISTANCE) / SUM (WEIGHT)
```

is evaluated. When control is returned, its value is passed back as the return value.

If a function is defined with a return mode, it may return control only by using a RETURN statement with a return-value expression. A simple RETURN (without an expression) is illegal, as is an implied return which occurs when control passes in sequence to the END of the FUNCTION body. Furthermore, the expression in the RETURN must be either of the return mode, or of a mode convertible to the return mode. A function that is defined to return a REAL, for instance, may not contain a RETURN statement that specifies a BOOLEAN return value.

We have now covered enough to be able to revise the center-of-gravity program in the desired manner. This is the result:

```
VARIABLES WT, DIS ARE ARRAY(25) REAL,
   N IS INTEGER;
REPEAT
   READ_LINE (N);
      IF N = 0 THEN TERMINATE;
      IF N < 0 | N > 25 THEN
         BEGIN;
            PRINT ('*****ERROR:  NUMBER OF WEIGHTS IS ' ||
               'NEGATIVE OR >25');
            TERMINATE;
         END;
```

```
      READ_LINE (WT(N AT 1), DIS(N AT 1));
      PRINT (WT(N AT 1), DIS(N AT 1),
         GRAVITY_CENTER (WT(N AT 1), DIS(N AT 1)));
END;

GRAVITY_CENTER:
   FUNCTION (WEIGHT, DISTANCE ARE ARRAY(*) REAL) REAL;
      IF SUM (WEIGHT) = 0 THEN
         BEGIN;
            PRINT ('*****ERROR:  WEIGHT SUM = 0');
            TERMINATE;
         END;
      RETURN SUM (WEIGHT * DISTANCE) / SUM (WEIGHT);
   END GRAVITY_CENTER;
```

Array return values of unresolved size

Size resolution of array return values may be specified in ways exactly analogous to those for size resolution of array parameters. That is, one or more of the dimension sizes following ARRAY in the return mode may be indicated by an expression containing a variable, in which case the size of the returned value may vary from call to call.

Alternatively, automatic size resolution of the return value may be indicated, by writing a * instead of one or more sizes. The size of the return value is then set to match the size of the expression in the RETURN statement that ends the function.

Automatic size resolution is especially important when you want the size of the return value to depend on the size of one of the parameters. One example is a function to sum over just the first dimension of a two-dimensional array. This function has a two-dimensional REAL parameter ARRAY_2, whose sizes are resolved to those of the argument. It finds the sum of all ARRAY_2(I), where I ranges from 1 to SIZE (ARRAY_2, 1). The size of the return value is always set equal to SIZE (ARRAY_2, 2).

```
SUM_2:
   FUNCTION (ARRAY_2 IS ARRAY(*,*) REAL COPYIN)
      ARRAY(*) REAL;
      FOR I FROM 2 THRU SIZE (ARRAY_2, 1) REPEAT
         ARRAY_2(1) := ARRAY_2(1) + ARRAY_2(I); END;
      RETURN ARRAY_2(1);
   END SUM_2;
```

We assume here that I is declared as a program-level variable. A variable is also needed to hold the partial sums. The partial sums are one-dimensional arrays, whose sizes vary from call to call (because of the unresolved sizes of ARRAY_2). Therefore, it is not possible to declare a program-level variable to hold the partial sums, because such variables must have their sizes fixed. Instead, we make ARRAY_2 a COPYIN parameter, and use ARRAY_2(1) for the partial sums. (There are some better solutions to these problems, which are explored in a later chapter.)

STRING return values of unresolved length

Since strings are basically arrays of characters, all the rules for unresolved-size array return values carry over naturally to unresolved-length STRING return values.

A good example is a replacement for the billing program's PROCESS_INPUT_NAME. We want it to take an argument of 20 characters, remove surrounding blanks, and return a string whose length is equal to the number of characters in the name. If the argument is all spaces, a single space is returned. We can call the new function NAME_CHECK:

```
NAME_CHECK:
   FUNCTION (INPUT IS STRING(20)) STRING(*);
      FOR FIRST THRU 20 UNTIL INPUT(FIRST) ~= ' ' REPEAT; END;
      IF INPUT(FIRST) = ' ' THEN          # RETURN A SPACE #
         RETURN ' ';
      FOR LAST FROM 20 BY -1 UNTIL INPUT(LAST) ~= ' '
         REPEAT; END;
      FOR I FROM FIRST THRU LAST          # CHECKING LOOP #
         REPEAT
            IF (INPUT(I) < 'A' | INPUT(I) > 'Z') &
               INPUT(I) ~= ' ' & INPUT(I) ~= '''' THEN
               TERMINATE_WITH_ERROR ('INVALID CHARACTER ' ||
                  'WITHIN A NAME');
         END;
      RETURN INPUT(LAST - FIRST + 1 AT FIRST); # RETURN NAME #
   END_NAME_CHECK;
```

The value returned by this function can be assigned back to NAME:

```
NAME := NAME_CHECK (NAME);
```

because its length is always less than or equal to NAME's length. Or the function may be called directly from a PRINT statement:

```
PRINT (ACCT_NO, NAME_CHECK (NAME), OLD_BAL, PAYMENT,
   PAST_DUE, INTEREST, PURCHASE, NEW_BAL);
```

Returning the empty string

There are certain cases where it is very useful to be able to have a function return a string of length zero -- a STRING-mode value that contains no characters. Such a STRING value is called the empty string. It is represented by a literal consisting of two apostrophes with no characters between them:

```
''
```

You can assign the empty string to a variable of mode STRING:

```
NAME := '';
```

in which case NAME is filled out entirely with spaces. The function NAME_CHECK of the previous section could be written to return the null string when it finds that its parameter contains nothing but spaces:

```
IF NAME(FIRST) = ' ' THEN RETURN '';
```

Then the assignment

```
NAME := NAME_CHECK (NAME)
```

would, as before, leave NAME unchanged if it were all spaces to begin with.

Returning the empty string is perhaps most valuable when the return value is an operand to a concatenation. Concatenating the empty string to any other string leaves that string unchanged. For instance, both of:

```
NAME || ''
'' || NAME
```

have the same value as NAME.

Imagine now that the data for each billing program account is accompanied by three input strings: a first name, a middle name, and a last name. We could read these strings into three variables called FIRST_NAME, MIDDLE_NAME, and LAST_NAME, apply NAME_CHECK to all three, and concatenate the results together to form NAME:

```
NAME := NAME_CHECK (FIRST_NAME) || ' ' ||
   NAME_CHECK (MIDDLE_NAME) || ' ' || NAME_CHECK (LAST_NAME);
```

If MIDDLE_NAME is all blanks, NAME_CHECK (MIDDLE_NAME) returns the empty string; NAME is assigned just the first and last names (separated by two blanks). If all three parts of the name are blank, NAME ends up all blank, as before.

Advanced topics

We have covered here just about everything that can be done with return values. The main exception is typified by SUM, which returns an INTEGER or a REAL depending on the mode of its argument. It is possible to define your own functions that return more than one possible mode. The mechanics of doing so are explained in the Primer's second volume.

20.0
PRE-DEFINED FUNCTIONS FOR ARRANGEMENT OF INPUT AND OUTPUT

The PRINT statement, as we have been using it, does not leave very much choice in the arrangement of output. Each output value is printed in the standard form for its mode -- and the standard form for REALs, in particular, is often cumbersome. When more than one numeric value is output, the values are all strung together, separated by single spaces; it's difficult or impossible to get values to line up in columns. The READ_LINE statement, too, has its limitation. Input items must be separated by blanks in order to be read as separate items.

What is needed are some ways of making the arrangement of items to be output and input more flexible. We need some way to specify what form a piece of data will take and how many spaces will appear on each side of it, in order that it be positioned properly on the page and be properly aligned with other items of its kind. On input, we need a way to separate items which were not separated by blanks on the cards and a way to specify which columns the pertinent data will appear in. The purpose of this chapter is to show you some ways of specifying those facts about data items.

Centering character strings on output

Let's take a simple sort of problem first. Suppose you want to print a heading, such as TABLE #1, and you want that heading to be centered in the middle of the printout paper. If you specify simply

```
PRINT ('TABLE #1');
```

the character string will be printed way over on the left side of the paper:

```
TABLE #1
```

One way to solve the problem would be to increase the length of the character string by adding enough blanks onto the left of the heading so that even when it begins at the left margin of the paper, the "visible" portion will be positioned correctly on the page, held in place by a carefully counted number of invisible blank character spaces of "padding":

```
PRINT ('                    TABLE #1');
```

This method works conveniently enough if the amount of "padding" to be added is small. But if the printout paper is 132 characters wide, there will not be enough spaces on a card to hold all of the blank characters between the opening apostrophe and the first letter of the title.

Fortunately, CS-4 has a pre-defined function which solves the problem by allowing you to specify the number of blank spaces which are to precede a given STRING. The name of the function is LPAD. The first argument of LPAD is the string to be "left padded", and the second argument is an integer. The return value of LPAD is of mode STRING, and its length is specified by the integer argument. The rightmost characters of that string are the characters of the string argument to LPAD, and the remaining characters to the left of it are space-characters. Applied to 'TABLE #1', the LPAD function can look like this:

```
LPAD ('TABLE #1', 70)
```

The return value of this function is a string 70 characters long. The first 62 characters are space-characters, and the remaining characters are the eight characters of 'TABLE #1'. When the LPAD function is used in the PRINT statement, like this,

```
PRINT (LPAD ('TABLE #1', 70));
```

the heading will be properly positioned in the center of the 132-character-wide printout paper. The string is still printed beginning at the left margin, but the first 62 characters of it are "invisible" space characters.

Arrangement of INTEGER output in columns

Let's review what the PRINT function does with integer arguments. Simply including a comma-separated list of data items in a PRINT statement like this:

```
PRINT (ACCT_NO, OLD_BAL, PAYMENT, PAST_DUE, INTEREST,
   PURCHASE, NEW_BAL);
```

will produce printed lines that look like this:

```
705 5950 2975 2975 45 9995 13015
7 0 0 0 0 13090 13090
26 -750 0 0 0 1250 500
0 0 0 0 0 0 0
```

Each successive execution of the PRINT statement, with new values for the INTEGER variables, produces a new line of printout. Each integer takes up only the space that is required in order to print it, with the larger integers taking up more space than the smaller ones. To keep the numbers from all running together, PRINT leaves one extra space after each number printed. But that one space separating each item does not provide easily readable columns of numbers. What we want is output that looks like this:

705	5950	2975	2975	45	9995	13015
7	0	0	0	0	13090	13090
26	-750	0	0	0	1250	500
0	0	0	0	0	0	0

We want each integer to be aligned with the one above it, "right justified" within a "field" whose width is independent of the size of the number at any given execution of the PRINT. Even when the value of the item is zero, we want enough blank characters "padded" on the left of it so that the zero is aligned with the right-most digit of the corresponding item above it. Whatever the size of the integer, we want the amount of padding to vary accordingly, so that the padding plus the size of the integer always fills up the total width of the column.

If we know, for example, that ACCT_NO will always be 999 or less, we want to be able to specify a constant column width of three for all ACCT_NOs. CS-4 permits the LPAD function to be used in situations like this. To specify that the ACCT_NO variable is to take up three character spaces, simply write

```
LPAD (ACCT_NO, 3)
```

and ACCT_NO will be padded on the left with enough space-characters to make it a total of three characters long. The LPAD function is defined in such a way that if its first argument is not a STRING already, but is one of the other basic modes in CS-4 (such as INTEGER, REAL, or BOOLEAN),

the value of the argument will be converted to STRING before the padding is done. Given the following values of ACCT_NO, here are the corresponding return values of LPAD (ACCT_NO, 3):

ACCT_NO	LPAD (ACCT_NO, 3)
705	'705'
7	' 7'
26	' 26'
0	' 0'

Notice that for ACCT_NO 705, no space-characters need to be added, since '705' is already of length 3.

The LPAD function with its integer argument may be placed directly in a PRINT statement, and the returned, padded STRING will be printed out just like any other STRING. (Apostrophes, of course, do not appear on the printout.) Since we want all of the items in the PRINT statement in the billing program to line up in columns, we may place each item in its own LPAD function, specifying for each the width that we desire its column to be. Remember that the return values will be printed as STRINGs, so PRINT will not leave an extra space after each item as it does when it prints INTEGERs. So we must make the second argument to each of the LPADs large enough to insure that even for the largest integer in a given column, there will always be at least one space separating it from the item on its left. The complete PRINT statement will look like this:

```
PRINT (LPAD (ACCT_NO, 3), LPAD (OLD_BAL, 8),
   LPAD (PAYMENT, 8), LPAD (PAST_DUE, 8), LPAD (INTEREST, 5),
   LPAD (PURCHASE,8), LPAD (NEW_BAL, 8));
```

and the output it produces will be in columns, like this:

```
705    5950    2975   2975   45    9995   13015
  7       0       0      0    0   13090   13090
 26    -750       0      0    0    1250     500
  0       0       0      0    0       0       0
```

Now let's consider another possible change we might want to make in the output. Suppose we want to left-pad ACCT_NO, not with blanks, but with zeros. If we know that all account numbers will be less than 1000,

how do we make the integers appear on the paper as ranging from 000 to 999? (When ACCT_NO is 7, we want to print 007; when it is 26, we want to print 026, etc.)

LPAD may be used to solve this problem, too. LPAD may be given an optional third argument, which is the character to be used as padding. It must be a single character (enclosed in apostrophes if it's a literal), and it is separated from the first two arguments by a comma, like this:

```
LPAD (ACCT_NO, 3, '0')
```

This function has as its return value a STRING of length 3, the last characters of which are the characters of ACCT_NO (which LPAD first converts from INTEGER to STRING), and the preceding characters on the left are '0' characters:

ACCT_NO	LPAD (ACCT_NO, 3, '0')
705	'705'
7	'007'
26	'026'
0	'000'

Notice that for ACCT_NO 705, no '0' characters need to be added, since '705' is already of length 3.

We are now ready to write the complete PRINT statement that will produce the desired lines of output:

```
PRINT (LPAD (ACCT_NO, 3, '0'), LPAD (OLD_BAL, 8),
   LPAD (PAYMENT, 8), LPAD (PAST_DUE, 8), LPAD (INTEREST, 5),
   LPAD (PURCHASE, 8), LPAD (NEW_BAL, 8));
```

The output will look like this:

```
705    5950    2975    2975    45    9995   13015
007       0       0       0     0   13090   13090
026    -750       0       0     0    1250     500
000       0       0       0     0       0       0
```

The third argument of LPAD, the character to be used for padding, is optional. If it is omitted, a ' ' character is supplied by default. (We could have, if we had wanted to, written LPAD (INTEREST, 5, ' ') instead of LPAD (INTEREST, 5). The returned string would have been the same.) The third argument may, of course, be any character, not just ' ' or '0'.

A few words of caution concerning the use of LPAD: first, make sure that the length of the item to be padded does not exceed the value of the integer expressing the length of the string to be returned. It would be an error to write LPAD (OLD_BAL, 3) and then send it an OLD_BAL whose value was 5950 or -750. Second, remember that LPAD produces a STRING value, regardless of the mode of the expression to be padded. If OLD_BAL is an integer, it is perfectly legal to add another integer to it:

```
OLD_BAL + 30
```

but it would be an error to write

```
LPAD (OLD_BAL, 8) + 30
```

because one cannot add STRINGs to INTEGERs.

Because LPAD returns a STRING value, it is permissible to use the string concatenation operator || to join LPAD-ed expressions together or to join them to other strings. It is even legal to use sub-string subscripting to extract a "slice" of a STRING returned from an LPAD function. The expression

```
LPAD ('JONES, BILL', 15, '*')(7 AT 1)
```

has the value '****JON'.

LPAD has an additional provision, which facilitates the printing of numbers with leading zeros. If the first argument of an LPAD function is INTEGER or REAL and is less than zero, and if the padding character is a '0', then the minus sign in the resulting string is moved to the left-most position. If NEW_BAL is -300, the function LPAD (NEW_BAL, 7, '0') returns the string '-000300' and not '000-300'. This provision permits you to use LPAD to print numbers with leading zeros without having to worry about embedding minus signs after leading zeros in the returned strings.

Using LPAD with arrayed variables

You will recall that if the argument of PRINT or READ_LINE is an arrayed item, the array is "unraveled" and the function behaves as if it had a whole list of arguments consisting of the unraveled elements of the array. LPAD has an analogous property with regard to arrays. If the first argument of an LPAD is an array, LPAD is executed for each of the unraveled elements. The return value of such an LPAD is not just a single string, but an array of strings, each padded according to the specifications given in the other argument(s). If VALUES is declared to be an ARRAY(5) INTEGER, then the call LPAD (VALUES, 8) returns a five-element array of STRINGs, each element of which is eight characters long. This array can then be used like any other array of length-eight STRINGs. It may be assigned to a variable of mode ARRAY(5) STRING(8), or it may be used as the argument of a PRINT statement:

```
PRINT (LPAD (VALUES, 8));
```

which, upon execution, will print the contents of each of the five character strings, one after another, on the same line:

```
  483641     289 1850399      15     270
```

The integers of VALUES, after being converted to STRINGs in LPAD, are right-justified in string fields of eight characters, with blanks used as padding. Each eight-column field is printed with no additional space after it, since what it printed is a series of character strings, and not a series of integers. Comma-separated STRING arguments in PRINT statements are printed juxtaposed on the paper, just as they would be if they were concatenated before being sent to PRINT. The statement

```
PRINT (LPAD (ACCT_NO, 3, '0'), LPAD (OLD_BAL, 8),
   LPAD (PAYMENT, 8), LPAD (PAST_DUE, 8), LPAD (INTEREST, 5),
   LPAD (PURCHASE, 8), LPAD (NEW_BAL, 8));
```

and the statement

```
PRINT (LPAD (ACCT_NO, 3, '0') || LPAD (OLD_BAL, 8) ||
   LPAD (PAYMENT, 8) || LPAD (PAST_DUE, 8) || LPAD (INTEREST, 5)
   || LPAD (PURCHASE, 8) || LPAD (NEW_BAL, 8));
```

are identical in their effect.

Arrangement of REAL output items

As we explained in an early chapter, there are two ways to represent REALs in CS-4. They can be written simply as a sequence of digits with a decimal point among them; or the digits may be followed by an exponent (the letter E followed by a number). A REAL appearing as an argument in a PRINT statement is always printed in the exponential form. Because that form is not very natural to the billing program, we have for the last several chapters been using integers instead of reals. In this section, we show how to use REAL values for the amounts in the billing program, and obtain the output in the more conventional decimal form. Printed with two digits to the right of the decimal point, they will be in the standard form used to represent dollars and cents.

Here is how the output from one execution of the PRINT statement looked in chapter six, using REALs instead of INTEGERs.

```
7.05000E+02 5.95000E+01 2.97500E+01 2.97500E+01 4.46250E-01
9.99500E+01 1.30146E+02
```

(Actually, all seven items would be printed on one line, if the line were long enough. Most printout paper is wider than this page.)

What is needed is a way to convert those reals from exponential representation to decimal form, with the ability to specify how many decimal digits are printed out. This may be accomplished with the predefined function DSTRING. DSTRING is a function which converts a REAL to STRING representation. The STRING will contain the REAL in decimal form.

DSTRING takes three arguments: the first is a REAL which is to be converted into a STRING. The second is an INTEGER giving the number of characters which precede the decimal point, and the third is an INTEGER giving the number of characters which follow the decimal point. If NEW_BAL is a REAL that we wish to represent in decimal instead of exponential form, we can specify six digits to precede the decimal point and two to follow the decimal point, by the following DSTRING function:

```
DSTRING (NEW_BAL, 6, 2)
```

Here are some values of DSTRING (NEW_BAL, 6, 2) corresponding to various values of NEW_BAL.

NEW_BAL	DSTRING (NEW_BAL, 6, 2)
1.30146E+02	' 130.15'
1.30902E+02	' 130.90'
5.00000E+00	' 5.00'
0.00000E+00	' 0.00'

(The apostrophes, again, aren't actually part of the returned value. The above examples show them merely to indicate where the strings begin and end.) As you can see, the length of the string produced by DSTRING depends only on the values of its second and third arguments; it is independent of the values of the REAL which is being converted. More specifically, if m and n are the second and third arguments respectively, the length of the resulting string is m + n + 1, where the (m + 1)st character is the decimal point. The second argument to DSTRING determines how many characters will appear to the left of the decimal point; this number must be at least large enough to contain the largest REAL to be sent to the function. (Any extra spaces on the left of the decimal point will be padded in with space-characters.) If the REAL is negative, a '-' character will be inserted adjacent to the left-most digit. If the REAL is between -1 and +1, a '0' character will appear in the position immediately to the left of the decimal.

The third argument determines how many character positions will appear to the right of the decimal point. (The REAL will be rounded to this number of places.) If needed to fill out the field specified by this third argument, '0' characters will be added.

We are now ready to revise our billing program, using REALs instead of INTEGERs for all of the values to be printed out (except for ACCT_NO, which will remain an integer). DSTRING function calls now replace the REALs in the PRINT statement. The new version of the PRINT statement looks like this:

```
PRINT (LPAD (ACCT_NO, 3, '0'), DSTRING (OLD_BAL, 6, 2),
   DSTRING (PAYMENT, 6, 2), DSTRING (PAST_DUE, 6, 2),
   DSTRING (INTEREST, 3, 2), DSTRING (PURCHASE, 6, 2),
   DSTRING (NEW_BAL, 6, 2));
```

The printout will be in columns, with decimal points aligned:

705	59.50	29.75	29.75	0.45	99.95	130.15
007	0.00	0.00	0.00	0.00	130.90	130.90
026	-7.50	0.00	0.00	0.00	12.50	5.00
000	0.00	0.00	0.00	0.00	0.00	0.00

To get more spaces between a given item and the one to its left, simply increase the size of the second argument of that item's DSTRING function. Remember that the return value of DSTRING is a STRING, not a REAL or INTEGER, so PRINT does not supply an additional space after DSTRINGs.

Care must be taken in the selection of values for the second and third arguments. If the third argument is too small, too much data will be lost when the REAL is rounded off. If the third argument is zero, the decimal point will be the last character in the string, and the number will be rounded to the nearest whole number.

It is even more important that the second argument of the DSTRING function be large enough. If that argument does not allow enough characters to the left of the decimal point to contain the largest REAL to be sent to that DSTRING, that function call is in error. (There are ways to "recover" from errors like this, but that's a subject for the advanced features volume of the Primer.) If the REAL is negative, you must allow enough space for a minus sign as well as the digits. It's a good idea to allow a few extra characters for "padding", to separate DSTRINGs from items on the left.

Printing REALs in exponential form

The default representation of REALs is in exponential form. If VALUE is a REAL equal to 2.07385E+03, the command PRINT (VALUE) will cause the entire REAL to be printed: 2.07385E+03. Sometimes, however, what is desired is a shortened form of the REAL, but still in exponential form: 2.074E+03, or 2.1E+3. There is a function in CS-4 which allows you to specify the number of digits of mantissa and the number of digits of exponent in a STRING representation of REAL. It is a function like DSTRING in that it converts the REAL to STRING representation. The function is ESTRING, and the parentheses following it require three arguments: the REAL, an INTEGER specifying the number of digits of mantissa you want to precede the 'E', and an INTEGER specifying the number of digits of

exponent that you want to follow the 'E'. If R is the REAL to be converted, a call to ESTRING specifying 3 digits of mantissa and 1 digit of exponent is written like this:

```
ESTRING (R, 3, 1)
```

Here are some examples of strings produced by that call, for various values of R:

R	ESTRING (R, 3, 1)
3.81752E+02	' 3.82E+2'
6.57831E+01	' 6.58E+1'
-7.61420E+01	'-7.61E+1'
9.75241E-01	' 9.75E-1'

The length of the returned string depends only on the values of the second and third arguments; it is independent of the value of the REAL. Specifically, if m is the second argument and n is the third argument, the return value of ESTRING is as follows: The first character is a space or a minus sign, depending on the sign of the REAL. The next character is the first digit of the mantissa, the third character is a decimal point, and then the remaining m-1 digits of mantissa follow. The remaining characters are an E followed by a + or - sign, and finally, n digits of exponent. The length of the returned string, then, is always equal to the second argument plus the third argument plus four.

As with DSTRING, you must choose the value of the second and third arguments carefully. With ESTRING, the value of the second and third arguments must both be 1 or more, or the function call is in error. If the second argument is less than the number of digits of mantissa held in the machine, the mantissa will be rounded off as required. If the value you specify is greater than the number of digits of mantissa held in the machine, zeros will be added as necessary. The third argument should almost always be 1 or 2, except for extremely large numbers. (Remember, it indicates how many digits of exponent you want included in the string, not what those digits are. If you specified ESTRING (R, 6, 5), and R is 3.81752E+04, the returned string would be ' 3.81752E+00004'.) The function call will be in error if the third argument is 1 and the REAL is so large that it requires a two-digit exponent.

Like the strings returned by LPAD, strings produced by DSTRING and ESTRING may be used in any expressions where STRINGs are permitted, assuming their lengths are appropriate. Substring subscripting may be used to extract a selected number of digits of the REAL (should you for some reason ever want to do so). If R is 3.81752E+04, the expression

```
ESTRING (R, 6, 1)(3 AT 6)
```

selects the character string'752'.

Like LPAD, the functions DSTRING and ESTRING may take an array as a first argument. (With DSTRING and ESTRING, though, this array must be an array of REALs.) If the first argument is arrayed, the function is performed for each of the items in the array. The results may then be sent to a PRINT statement where they will be treated just like a comma-separated list of items. For example, if VALUES is an ARRAY(5) REAL, of which VALUES(1) is 3.81752E+04, VALUES(2) is 6.58317E+01, VALUES(3) is 7.61420E+01, VALUES(4) is 1.56001E-01, and VALUES(5) is 9.75241E+00, the statement

```
PRINT (VALUES);
```

causes the following line to be printed:

```
3.81752E+04 6.58317E+01 7.61420E+01 1.56001E-01 9.75241E+00
```

The statement

```
PRINT (DSTRING (VALUES, 5, 2));
```

causes the following line to be printed:

```
38175.20   65.83   76.14    0.16    9.75
```

Similarly, the statement

```
PRINT (ESTRING (VALUES, 4, 1));
```

causes the following line to be printed:

```
3.818E+4 6.583E+1 7.614E+1 1.560E-1 9.752E+0
```

Remember that when the first argument to ESTRING is negative, the returned string does not begin with a space-character. When printing such an ESTRING, therefore, make sure that at least one space separates the ESTRING from the item on its left. If the previous item in the PRINT function is numeric, that space will be supplied automatically. But if the previous item is a string, you must insert a space-character argument in the PRINT command, or LPAD the entire ESTRING function.

A programmer does not need to know in advance how many elements an arrayed first argument to an LPAD, DSTRING, or ESTRING function will have. If, for example, WEIGHT is an array of REALs, and N may vary from 1 through 10, it is possible to write

```
PRINT (N, DSTRING (WEIGHT(N AT 1), 4, 2));
```

and the DSTRING will be executed N times, and the N strings that are returned will be printed out, one after another, on the same line, following the integer N.

Printing on the same line twice

Sometimes it is advantageous to "overprint" on the same line. Overprinting is occasionally used in printing graphs, where one line of output needs to be superimposed on another. More often it is used to print underscores for lines of words. CS-4 has a statement which enables one to print over what has already been printed; this is the function OVER_PRINT. OVER_PRINT takes arguments and behaves in exactly the same way that PRINT does, except it does not advance the paper to a new line. So whatever OVER_PRINT puts out gets printed on top of whatever was printed on the previous line. For example, the statements

```
PRINT ('THE VALUES FOR TEMPERATURE RANGE ARE:');
OVER_PRINT (LPAD (LPAD ('_', 17, '_'), 32));
```

cause the following line to be printed:

THE VALUES FOR TEMPERATURE RANGE ARE:

The PRINT statement causes the string of English words to be printed on one line (actually, the command PRINT advances the paper to a new line, and then the string is printed). Then the OVER_PRINT statement

is evaluated and executed. The inner LPAD expands the string '_' by padding 16 additional '_' characters on to the left of it. Then the result of that padding operation is used as a string argument to the outer LPAD function, which pads enough blank spaces onto the left of it so that the final string is 32 characters long. The first 15 characters of the completed string are space characters, and the last 17 characters are '_' characters from the inner LPAD function. That long string is then printed out, superimposed on the previously printed line. The 15 space characters padded onto the left of the underscores are just enough to position the underscores under the proper words.

Advancing the printer to a new page

Suppose you want only 50 lines of numeric output from the billing program on a page (there is not room for much more, in any case). You may force the printer to advance to a new page by a command called PAGER, which can take arguments just like PRINT and OVER_PRINT do. PAGER simply directs the printer to advance to the beginning of a new page before any items in the argument are printed out.

We may use PAGER to have the headings of the various columns of billing program figures printed out at the top of each page. We may also put in a page number -- whose value is maintained in an INTEGER variable PAGES. The statement

```
PAGER ('ACCT #  OLD BAL.  PAYMENT  PAST DUE  INTEREST  ',
   'PURCHASE  NEW BAL.', LPAD ('PAGE ', 60), PAGES);
```

advances the paper to a new page and then prints the column headings. And, toward the right-hand side of the same line (assuming the paper is 132 character-spaces wide), it prints the word PAGE and the page number. (Of course, now that we have column headings for the values in the billing program, it will be necessary to select values for LPAD and DSTRING arguments in the PRINT statement so that the various numbers will line up neatly under the proper headings.)

The above PAGER command may be added to the billing program in a new function we may define and name START_NEW_PAGE. If the number of lines printed is stored in an INTEGER variable LINE_COUNT, we may define START_NEW_PAGE as follows:

```
START_NEW_PAGE:
   FUNCTION;
      PAGES := PAGES + 1;
      PAGER ('ACCT #  OLD BAL.  PAYMENT  PAST DUE  INTEREST  ',
         'PURCHASE  NEW BAL.', LPAD ('PAGE ', 60), PAGES);
      PRINT;
      LINE_COUNT := 1;
   END START_NEW_PAGE;
```

The function increments the page number, turns the page, and prints out the column headings and the new page number. Then, just before the function returns, it resets LINE_COUNT to one, so that we are ready to begin the line counting for the new page.

At program level, we will need a statement which increments LINE_COUNT whenever a line is printed. Then, to test whether START_NEW_PAGE should be invoked, we execute a statement such as:

```
IF LINE_COUNT >=51 THEN START_NEW_PAGE;
```

which calls START_NEW_PAGE if 50 lines have been printed.

Printing multi-dimensioned arrays as tables

The pre-defined functions explained in the preceding sections may be used in various combinations in order to print arrayed quantities in whatever format you desire. This allows you to output values in a form that is much more easily readable than is possible with the default output arrangement. Consider, for example, the output from the temperature grid problem of chapter 13, where TEMPERATURE is an ARRAY(9,6) REAL. The default output arrangment provided by PRINT (TEMPERATURE) produces an unraveled list of the 54 elements, printed one after another, beginning on a new line only when one line is filled. The first few lines of such output might look like this:

```
1.34254E+02 1.34778E+02 1.34671E+02 1.34698E+02 1.34730E+02
1.34053E+02 1.33732E+02 1.24907E+02 1.16734E+02 1.16854E+02
1.28236E+02   . . .
```

The values appear neatly aligned, but that is just because the default representation of REALs has a consistent number of digits placed in each positive number. A new line is begun only when a previous line is filled, and that does not necessarily occur when all the items in one dimension of the array are printed. What we want is a table with nine rows of six columns each.

We may use PAGER, OVER_PRINT, LPAD, and DSTRING not only to produce the desired table, but to include whatever column and row headings we wish. TEMPERATURE is the (9,6) REAL of the temperature-grid problem that we mentioned earlier, and I is an INTEGER control variable. (In order that all 54 values not be identical, let's assume the values in the array represent temperatures on the sheet of metal at some time before equilibrium is reached.)

The following statements will produce the desired table:

```
PAGER (LPAD ('TABLE #1', 32));
PRINT (LPAD ('TEMPERATURE GRID', 36));
OVER_PRINT (LPAD (LPAD ('_', 16, '_'), 36));
PRINT;
PRINT ('       |  COL. 1  COL. 2  COL. 3  COL. 4  COL.5  COL.6');
OVER_PRINT (LPAD ('_', 56, '_'));
FOR I THRU 9
   REPEAT
      PRINT ('       |');
      PRINT ('ROW ', I,  ' |', DSTRING (TEMPERATURE(I), 5, 2));
   END;
```

The table will be printed at the top of a new page:

TABLE #1

TEMPERATURE GRID

	COL. 1	COL. 2	COL. 3	COL. 4	COL. 5	COL. 6
ROW 1	134.25	134.78	134.67	134.70	134.73	134.05
ROW 2	133.73	124.91	116.73	116.85	128.24	136.32
ROW 3	133.68	113.70	96.32	98.75	119.76	139.02
ROW 4	133.62	105.82	75.42	76.38	109.40	140.72
ROW 5	133.55	102.67	68.52	69.70	105.08	140.97
ROW 6	133.48	106.72	80.62	81.37	112.60	140.58
ROW 7	133.40	117.85	101.69	102.83	120.03	140.16
ROW 8	132.31	125.21	117.54	118.27	127.41	139.06
ROW 9	132.18	133.02	134.15	135.20	136.96	138.15

Since TEMPERATURE is a two-dimensional array, TEMPERATURE(I) is a one-dimensional "slice" (one row) of that array. Therefore, the PRINT statement with TEMPERATURE(I) in the REPEAT loop causes the whole row of temperatures to be printed out, each appropriately in decimal form with five characters (including spaces) on the left side of the decimal point, and two on the right. An extra PRINT statement in the REPEAT loop makes for double-spacing between rows.

Using variables as second arguments in LPAD functions

LPAD has uses other than right-justifying columns of figures. We have seen some of these in the previous example of the table of temperatures. Another type of display can be created by using an INTEGER variable as the second argument of LPAD.

Suppose a program were devised that keeps track of the number of days during which the outdoor temperature falls within certain ranges. If we are interested, say, in the temperatures for April and May, we may use an array declared as follows:

```
VARIABLE AM_TEMP IS ARRAY(100) INTEGER;
```

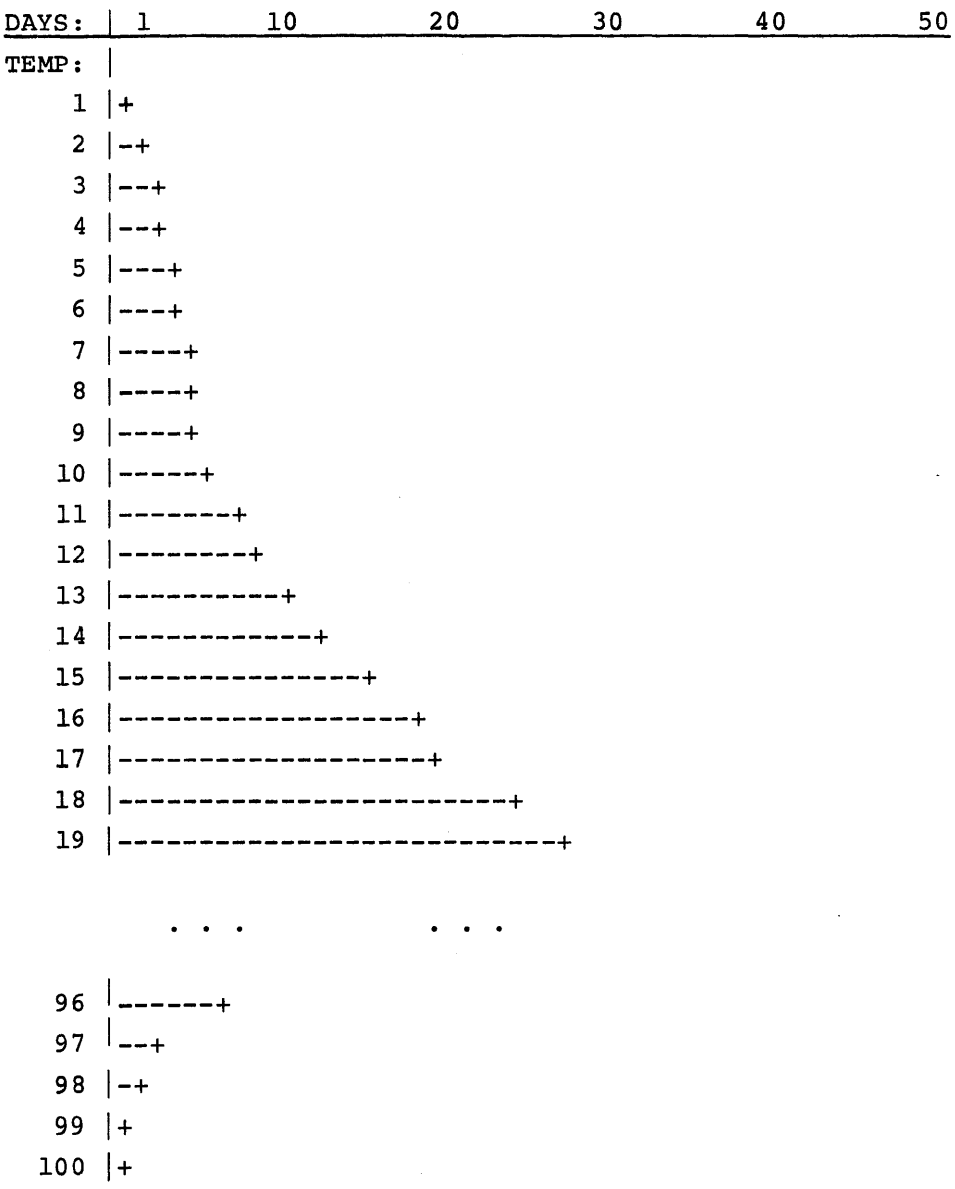
to count the days during which the temperature falls within each degree range between 1 and 100 degrees above zero. For every day during which the temperature falls within zero and one degree above zero, AM_TEMP(1) is increased by one; every day during which the temperature gets between eight and nine degrees above zero, AM_TEMP(9) is increased by one, etc. (The details of how the program is written or how the temperatures are recorded need not concern us -- we are interested only in how to arrange the printed output.)

By using LPAD to display the contents of this array, it is possible to construct a horizontal bar graph to show the relative frequencies of occurrence of the various temperatures. The following statements will produce the graph:

```
PAGER (LPAD ('TEMPERATURE CURVE FOR APRIL - MAY', 45));
OVER_PRINT (LPAD (LPAD ('_', 33, '_'), 45));
PRINT;
PRINT ('DAYS: | 1        10         20         30',
   '         40         50');
OVER_PRINT (LPAD ('_', 58, '_'));
PRINT ('TEMP: |');
FOR I THRU 100
   REPEAT
      PRINT (LPAD (I, 5), ' |', LPAD ('+', AM_TEMP(I) + 1, '-'));
   END;
```

The following graph will be printed, beginning at the top of a new page:

TEMPERATURE CURVE FOR APRIL - MAY



The column of figures on the left contains the subscripts of AM_TEMP, and the length of the line of dashes for each subscript represents the number of days during the two-month period during which the temperature was within the range of that subscript and the one above it. LPAD places that many dash characters on the left of the '+' character. (The '+' was chosen to make the right-hand end of the horizontal line more definitely marked.) What makes the graph is that each time the REPEAT statement loops, a new line of the graph is printed. Its length is determined by the magnitude of the value of that particular element in the array. Of course, with 100 subscripted values, the whole graph will not fit on one page of printout. The graph may be split onto two pages, or it may be shortened by adding adjacent pairs of elements together and printing a "compressed" graph.

In constructing this kind of display, make sure that the second argument of the LPAD function always evaluates to an INTEGER that is at least as large as the length of the item to be left-padded. (In this example, 1 was added to AM_TEMP(I) to cover those cases in which the number of days was zero.) Also, make sure that the magnitude of the various elements does not exceed what can be printed on one line.

Input of data items using READ_LINE

Fortunately, you do not need to worry about arrangement of data items as much on input as on output. On output, alignment with LPAD, DSTRING, or ESTRING was often necessary in order that the proper number of spaces separate the various columns of data items on paper. But on input, the number of blanks separating numeric and BOOLEAN data items is irrelevant. As long as they are separated by at least one blank, they are assigned in the order that they appear to the respective "target variable" arguments of READ_LINE.

For the CS-4 modes that are used in this volume of the Primer, PRINT and READ_LINE are "compatible" -- that is, whatever variables appear in a PRINT function can later appear in a READ_LINE function, in the same order, and whatever form the values were printed in will be acceptable for variables of that same mode on input. That is why PRINT inserts a blank space after each numeric and BOOLEAN item printed. That way, we are guaranteed that at least one space will be present to assure that READ_LINE will be able to separate adjacent data items.

(There is a way to read data items that are not separated by any blanks at all; how to do that will be the subject of a subsequent section in this chapter.)

INTEGERs printed with the aid of LPAD and REALs printed with DSTRING or ESTRING can, of course, be read with READ_LINE as numerics (INTEGERs or REALs), even though they technically were STRINGs when they were printed. If NAME is a STRING(20) variable, AGE is INTEGER, and WAGES is REAL, the statement

```
PRINT (NAME, LPAD (AGE, 3), DSTRING (WAGES, 3, 2));
```

will produce lines of output that look like this:

```
KATHERINE M. O'NEILL 42  4.25
CHRISTIANSEN JONES   35  5.80
JOEL MILLER          19  1.25
```

Assuming that the above output is punched on cards, the output could be read as input with the statement

```
READ_LINE (NAME, AGE, WAGES);
```

where NAME is STRING(20), AGE is INTEGER, and WAGES is REAL. (In volume 2 of this Primer we'll show how to get the computer to punch output on cards instead of printing it on paper.) READ_LINE assigns to NAME the contents of the first 20 characters (whatever they are -- blanks included). Then it assigns the next data item to AGE and skips a space (just as PRINT would have done had it printed AGE as an INTEGER without LPAD). Then it does the same for WAGES. Like PRINT, READ_LINE skips a space after numbers, but not after STRINGs. Therefore, no space is mandatory between the NAME data item and AGE. (In this case, AGE may begin in column 21.) A space is, however, necessary between AGE and WAGES, since both are read as numerics. Had AGE and WAGES not been converted to STRING through LPAD and DSTRING prior to being printed, that space would have been inserted automatically. But since they were printed as STRINGs, it is up to the programmer to insure that enough padding is allowed that a space will remain on even the largest data items. As you can see, this is especially important if the output data are to be subsequently read with

READ_LINE. Of course, any number of blanks, may appear between any of these items.

The fact that READ_LINE skips a space after numerics is particularly important when STRINGs follow numeric items. If AGE is INTEGER and NAME is STRING(20), the statement

```
READ_LINE (AGE, NAME);
```

unlike the READ_LINE on the previous page, would require that AGE and NAME be separated by a blank. If the right-most digit of AGE is in column 6, READ_LINE (AGE, NAME) would assign columns 8 through 27 to NAME.

There is another precaution that you should take when reading REAL values with READ_LINE. You will recall that each REAL is stored in the computer with six digits of precision. (There is a way to vary that degree of precision; it will be explained in volume 2 of the Primer.) We just discussed in the previous section that it is possible to print out fewer than six digits of precision by using DSTRING or ESTRING. What happens if output from a CS-4 program using DSTRING is used as input to another program? If a variable R in a program has the value 2.58385E+02 and is printed with DSTRING (R, 3, 1), and the value that is printed out (258.4) is read into variable X in another program with READ_LINE, the digits of precision that are lost with DSTRING will be made up with zeros. X in the second program will not be 2.5838E+02, but 2.58400E+02. This loss of precision varies according to the amount of rounding that takes place in DSTRING or ESTRING. So, if

```
PRINT (DSTRING (X, 4, 2));
```

produced these lines of output:

```
7430.21
 386.67
  64.83
   0.07
```

there was no loss of precision in producing 7430.21, for all six digits of precision were printed with DSTRING. However, it is quite possible that 0.07 was a REAL with exponential representation of 6.63719E-02. All but one of the significant digits of precision were lost in the DSTRING print statement. If this number were read with READ_LINE, 0.07 would be

assigned to a REAL as 7.00000E-02. These facts should be kept in mind when printing out REALs. The ease in readability of evenly aligned decimals might in certain cases be offset by the loss of precision, especially when the values vary over a wide range. The loss of precision would have been less, or at least more uniform, had the values been printed with ESTRING.

Input of data items not separated by blanks

Suppose that each card to be read has punched on it a post office zip code number and the number of pieces of mail postmarked at that office in one day. The first four cards might look like this:

```
63034  15632
63012   1472
63054   7915
63082  13901
```

Suppose further that we want to split apart the zip code number and assign the first three digits of it (which indicate the postal zone) to an integer variable ZONE_NO and the last two digits of it (which indicate the particular post office building) to an integer variable PO_NO.

The READ_LINE statement, however, treats the zip code number as one data item, not two. If we tried to read this data with the statement

```
READ_LINE (ZONE_NO, PO_NO, PIECES);
```

the entire zip code on the first card, 63034, would be assigned to ZONE_NO, and the other number would be assigned to PO_NO. PIECES would have no value from that card to be assigned to it at all. (READ_LINE would then read the second card and assign the zip code number on that card to PIECES.)

Run-together items like this may be assigned to the proper variables by first assigning the item to be sub-divided to a STRING variable, and then using sub-string subscripting to divide it into the proper data items. (In this example, the first three characters of the zip code will be the zone number and the remaining two will be the post office number.)

So, if we declare

```
VARIABLE ZIP_CODE IS STRING(5);
```

then the command

```
READ_LINE (ZIP_CODE, PIECES);
```

reads in a card of the above data. Assuming that the zip code number begins in column one, ZIP_CODE(3 AT 1) may then be assigned to a variable ZONE_NO.

One problem still remains, though, before we can make that assignment. ZIP_CODE(3 AT 1) is still of mode STRING, and we want ZONE_NO to be an INTEGER. Earlier, we learned that integers were converted to strings (for use in LPAD) "implicitly" -- that is, the compiler performed the conversion for us when it found the integer where it "expected" to find a string. The other basic modes in CS-4 (such as REAL and BOOLEAN) convert to STRING in the same way. But now we want to make the conversion in the other direction -- we want to convert string-to-integer, not vice-versa. This kind of conversion does not take place implicitly in CS-4. Instead, there are special pre-defined functions that you must use if you wish to convert '630' to 630. The STRING-to-INTEGER conversion function is S2I (which is short for STRING-to-INTEGER), and it takes one argument, the string to be converted. The return value is the integer that would result if the apostrophes were removed from the string-literal. Thus,

```
S2I ('630')
```

evaluates to the integer 630.

The argument of S2I must contain a sequence of digits. Space-characters and/or a sign may precede the digits in the string, and space-characters may follow it, but no other characters are permitted.

We are now ready to read the cards with the run-together data. Here are the cards, with the data punched beginning in column one:

```
63034  15632
63012   1472
63054   7915
63082  13901
```

In order to divide the first data item and make the conversions and assignments, these declarations are necessary:

```
VARIABLE ZIP_CODE IS STRING(5),
         ZONE_NO, PO_NO, PIECES ARE INTEGER;
```

These statements will produce the desired assignments:

```
READ_LINE (ZIP_CODE, PIECES);
ZONE_NO := S2I (ZIP_CODE(3 AT 1));
PO_NO := S2I (ZIP_CODE(2 AT 4));
```

Sometimes it is necessary to read run-together data that were prepared by persons who left out blanks in an attempt to squeeze as much data as possible onto each card. Because reading run-together data requires the round-about method of assignment to a STRING variable and then conversion from sub-strings to the desired target mode, most programmers prefer to have each data item on the input cards separated by one or more blanks. Blanks also make it easier to sight-check the cards for accuracy. CS-4 encourages the separation of data items by supplying a blank space automatically after each numeric and BOOLEAN argument in the PRINT function. (It is possible to run CS-4 output together, by converting the data items to STRINGs before printing. But the practice is not recommended.)

CS-4 provides conversion functions from STRING to other basic modes as well, which you can use in case you need to separate run-together data other than integers. The STRING-to-REAL function is

```
S2R (S);
```

where S is a STRING. S may consist of characters constituting either exponential or real-without-exponent representation. No blanks may appear between the first digit of the mantissa and the last digit of the exponent. (Leading and trailing blanks are permitted; these will be removed before the conversion is performed.) The following character strings are legal arguments to S2R, and they illustrate the kind of character strings that may be converted to REALs.

S	S2R (S)
'2.5'	2.50000E+00
'- .5'	-5.00000E-01
' + .2'	2.00000E-01
'2.5E4'	2.50000E+04
'-.25E-4 '	-2.50000E-05
'50.'	5.00000E+01
'1.5237496714E+02'	1.52375E+02

The following character strings, though, cannot be converted to REALs using S2R. Attempting to use them as the argument to S2R will result in an error.

ILLEGAL
'. 5'
'.2.'
'2.50E.2'
'.TEN'
'50'
'.E+02'
'2. E02'
'2 / 100'

These numerous examples are given not to encourage you to use "unusual" representations of REALs in STRINGs, but merely to show what the requirements for the conversion function are. Actually, these rules are identical to the rules for preparing REAL data for input to READ_LINE, except that here a sign may "float" anywhere in the field of leading blanks.

One other basic mode is used in this volume of the Primer: BOOLEAN. It is possible to convert from STRING to BOOLEAN with the function

S2B (S)

After leading or trailing blanks are removed from S, S converts to TRUE if S consists of 'TRUE' or 'T', or it converts to FALSE if S consists of 'FALSE' or 'F'. All other contents of S will be in error.

You may have noticed that STRING-to-INTEGER, -REAL, and -BOOLEAN conversions place rather strict stipulations as to which sequences of characters are legal and which are not. One reason for not being more

liberal and allowing wider variations in legal strings is that this strictness permits READ_LINE and the conversion functions to check for errors in the input data. Furthermore, if you wish to use STRING input and substring subscripting prior to conversion to other modes, you may make the additional check that the data always appear in the proper columns. Error checking like this protects programs from some common errors in data preparation as well as from your using the wrong input data.

Irrelevant input data

If the cards you wish to use for input were punched for other uses in addition to yours, you may not wish to use all the data on the cards. The only items you are interested in reading may be punched say, in columns 41 through 80. Columns 1 through 40 may contain irrelevant material. The easiest way to read the cards and get the data you are interested in, but "throw away" the "garbage", is to assign those columns you wish to disregard to a "dummy" STRING of that length. If, for example, ACCT_NO, OLD_BAL, PAYMENT, and PURCHASE were in columns 41 through 80, and any number of irrelevant items of whatever sort were punched in the first 40 columns, the pertinent data could be read as follows:

```
VARIABLE DUMMY IS STRING(40),
         ACCT_NO IS INTEGER,
         OLD_BAL, PAYMENT, PURCHASE ARE REAL;
READ_LINE (DUMMY, ACCT_NO, OLD_BAL, PAYMENT, PURCHASE);
```

The contents of columns 1 through 40 are now in DUMMY, but you need never use it. ACCT_NO may begin in col. 41; it need not be preceded by a blank, because READ_LINE does not skip a space after STRINGs.

If you need to make more than one "skip" per card, you can make the READ_LINE statement easier for you and other programmers to understand by making the dummy string of length 80:

```
VARIABLE DUMMY IS STRING(80);
```

and using subscripting to keep track of the columns to be skipped:

```
READ_LINE (DUMMY(20 AT 1), ACCT_NO, OLD_BAL,
           DUMMY(16 AT 45), PURCHASE, NEW_BAL);
```

You can skip a whole card in the input file by executing READ_LINE with no arguments. The statement

```
READ_LINE;
```

reads a card and "throws it away", regardless of its contents. You are then ready to read the next card.

21.0
INTERNAL NAMES AND STORAGE TYPES

A substantial part of our discussion of functions has been concerned with their use of variable and constant names. We can separate the names that appear in functions into three informal classes, according to how they are employed:

1) Names external to the function -- ones also referred to at program level or in other functions; we have been calling them shared names. They are used for data that must be known or manipulated in several different parts of the program.

2) Parameter names -- ones declared in function headings; they can be referred to only within the function that declares them. But they are used to pass data in or out of the function, to another part of the program.

3) Names local to the function -- ones which are referred to only within a function, and which contain data that is used only within that function. These include names of FOR variables needed only locally, and names of other variables needed to hold temporary results such as partial sums.

We have dealt with the first two classes fairly thoroughly, while the third class has been largely ignored. Names local to our sample functions have all been defined at program level, in the same way as external names; the only difference was that local names just happened to be used solely within a particular FUNCTION. This chapter shows how local names can be distinguished from external names, by declaring them within the body of the FUNCTION itself. We also explain how the programmer can decide in what way storage is to be set aside for local variables and constants. Finally, the similar properties of names declared within BEGIN blocks are described.

Toward the end of the chapter, we take up a related topic: the definition of functions within other functions, or within BEGIN blocks. This will set the stage for the following chapter, which is a general discussion of the recognition of names in a structured program.

Declarations inside a FUNCTION definition

As a first example, consider the billing program function STORE_FREE_ACCT_NUMBERS. Our last version of it was on page 190, Chapter 17. It contains three variables -- N, I, and FREE_ACCT -- which are just temporary storage for its own purposes and which are used nowhere else in the program. On the other hand its remaining variable, FREE_ACCT_NO, is not just used internally; it is shared with PROCESS_NEXT_ACCOUNT.

The three variables which are used only locally may be declared within the function body itself, instead of at program level. Normally they are placed right after the heading:

```
STORE_FREE_ACCT_NUMBERS:
   FUNCTION;
      VARIABLES N, I, FREE_ACCT ARE INTEGER;
      READ_LINE (N);
      IF N < 0 | N > 999 THEN
         TERMINATE_WITH_ERROR ('NUMBER of FREE ACCOUNTS ' ||
            'NEGATIVE OR >999');
      FOR I THRU N
         REPEAT
            READ_LINE (FREE_ACCT);
            IF FREE_ACCT < 1 | FREE_ACCT > 999 THEN
               TERMINATE_WITH_ERROR ('FREE ACCOUNT NUMBER IS ' ||
                  '<1 OR >999');
            FREE_ACCT_TABLE(FREE_ACCT) := TRUE;
         END;
   END STORE_FREE_ACCT_NUMBERS;
```

Declarations such as these are said to be internal to the function. One advantage of internal declarations is immediately apparent here: they make

a function definition easier to read. Variables and constants internal to a function are declared right where they are used, rather than at the top of the program -- where they would be mixed in with other names used by other functions.

When a variable or constant is declared internal to a function, it cannot be used outside of the function. This results in a subtle form of error-checking. If FREE_ACCT were accidentally written, say, instead of ACCT_NO in the function PROCESS_NEXT_ACCOUNT, the compiler would catch the error -- because FREE_ACCT is only declared for use within STORE_FREE_ACCT_NUMBERS.

Storage types

When we first introduced the concept of variable, we said that each variable had associated with it an area of storage in the computer. Constants, too, may have storage areas. The compiler makes sure that the storage space given to a variable or constant will be sufficient to hold a representation of whatever value the variable or constant might have.

The compiler does not itself set aside, or allocate, storage for variables or constants. Storage space is only allocated when it is needed -- when the program is run. Thus what the compiler actually does is to translate the program in such a way that the program, as it is running, can allocate space in the machine for itself when it needs it.

Program-level variables or constants can contain values used throughout the program. Therefore, the storage for names in program-level declarations is allocated as soon as execution of the program commences -- before control is passed to any of the program-level executable statements. Furthermore, the storage for these names is not relinquished, or freed, until the program's execution is concluded. (When a program is done it has to give up the space it is using, so that the space can be used by other programs.)

Storage allocation of this sort -- where allocation is the first act of the program, and freeing the last -- is only one of the storage types (or storage classes) in CS-4. The term storage type (or class) is always used to refer to the way storage is allocated -- not to where or how it is actually arranged in the computer. Each storage type has a name, so that it is possible to refer to it in a CS-4 program. All program-level variables, for instance, are of the type STATIC. STATIC is the default type for program-

level declarations. But if present, it is placed after the mode name:

```
VARIABLE ERROR IS BOOLEAN STATIC,
   FREE_ACCT_TABLE IS ARRAY(999) BOOLEAN STATIC;
VARIABLE PAGES IS INTEGER STATIC ::= 0;
```

Storage allocation for parameter variables depends on their type. Some parameters have their own space allocated, while others share storage with their corresponding arguments. But whatever the case, storage is never allocated for a parameter until the function that declared the parameter is called; and storage for a parameter is freed when the function returns. Thus when a function finishes, the space used by its parameters is relinquished, and it may later be used by other functions' parameters.

This treatment of parameter storage casts some light on why array parameters may have different sizes at different calls. The amount of storage an array takes up depends on its size. Since storage for a parameter is not allocated until the function is called, the size of an array parameter need not be determined until the function is called. And since space has to be allocated again each time the function is called (because it is freed each time the function returns), the size of the array can vary from call to call.

Variables declared internal to a function are treated very much like parameters. Their storage areas are allocated when the function is called, and freed when it returns. This storage type has the name AUTOMATIC. It is the default type for variables declared internal to a function, but it may be written in declarations just like STATIC:

```
VARIABLES N, I, FREE_ACCT ARE INTEGER AUTOMATIC;
```

Program-level variables must be STATIC, so writing AUTOMATIC in a program-level declaration is an error.

The position of a declaration within a function body does not affect when the storage is allocated. Even if an AUTOMATIC variable is

declared just before the END statement, its storage is allocated before any statement of the body is executed. The variable may thus be used in an executable statement that precedes its declaration statement. Similarly, program-level STATIC variables may be declared anywhere at the program level, but their storage is always allocated at the very start of execution.

Initialization of AUTOMATIC storage

If an AUTOMATIC variable is declared with an initialization expression, it is assigned that initial value each time it is allocated. For instance, in this function that takes an INTEGER value and returns its factorial:

```
FACTORIAL:
   FUNCTION (N IS INTEGER) INTEGER;
      VARIABLE MULT IS INTEGER,
         FACT IS INTEGER ::= 1;
      FOR MULT FROM 2 THRU N REPEAT
         FACT := MULT * FACT; END;
      RETURN FACT;
   END FACTORIAL;
```

the local variable FACT is initialized to 1 each time the function is called.

The initialization expression need not be just a literal. It may contain variables (or constants) external to the function, or parameters. Thus a local variable may be initialized to different values at different calls.

All the above applies with equal force to CONSTANT declarations. One consequence of this is that a constant local to a function may be declared to have different values at different calls. For example:

```
GRAVITY_CENTER:
   FUNCTION (WEIGHT, DISTANCE ARE ARRAY(*) REAL) ARRAY(*) REAL;
      CONSTANT NUMBER IS INTEGER ::= SIZE (WEIGHT);
```

Here on each allocation of NUMBER it is given a value equal to the size of WEIGHT. NUMBER may thus be allocated with several different values during the same execution of the program. But since NUMBER is a constant, it may not be assigned a new value during execution of GRAVITY_CENTER.

AUTOMATIC arrays and strings of Unresolved size

An array (or string) which is AUTOMATIC need not have the same dimension sizes at each allocation. One or more dimensions may be varied, by writing their sizes as expressions containing parameters or external variables. This feature can be put to use in the function SUM_2 of the previous chapter, which can use a local variable to store its partial sums:

```
SUM_2:
   FUNCTION (ARRAY_2 IS ARRAY(*,*) REAL) ARRAY(*) REAL;
      VARIABLE I IS INTEGER,
         ARRAY_SUM IS ARRAY(SIZE (ARRAY_2, 2)) REAL ::= 0;
      FOR I THRU SIZE (ARRAY_2, 1) REPEAT
         ARRAY_SUM := ARRAY_SUM + ARRAY_2(I); END;
      RETURN ARRAY_SUM;
   END SUM_2;
```

Each time this function is called, ARRAY_SUM is allocated with a size equal to that of ARRAY_2's second dimension, and all its elements are initialized to 0.

When an arrayed variable or constant is AUTOMATIC and has an initialization expression, a * may replace the size expression for one or more dimensions. These dimensions' sizes will then automatically be set to the sizes of the corresponding dimensions in the initialization expression. As a one-dimensional example, we could write SUM_2 with ARRAY_SUM initialized to the first element of ARRAY_2:

```
SUM_2:
   FUNCTION (ARRAY_2 IS ARRAY(*,*) REAL) ARRAY(*) REAL;
      VARIABLE I IS INTEGER,
         ARRAY_SUM IS ARRAY(*) REAL ::= ARRAY_2(1); END;
      FOR I FROM 2 THRU SIZE (ARRAY_2, 1) REPEAT
         ARRAY_SUM := ARRAY_SUM + ARRAY_2(I);
      RETURN ARRAY_SUM;
   END SUM_2;
```

ARRAY_SUM is here automatically allocated with a size equal to that of ARRAY_2(1). This version is a bit more efficient than the previous one, since the summing loop starts with I equal to 2 instead of 1.

STATIC variables and constants internal to a function

A variable internal to a function may be declared with the STATIC storage type. Because the default for internal variables is AUTOMATIC, the name STATIC must be written explicitly in the declaration. Like all internal variables, STATIC ones cannot be used outside the function where they are declared. However, storage for an internal STATIC variable is only allocated once -- when execution of the program that contains the function begins -- and its storage is not freed until the program is terminated. Thus the value of a STATIC variable is not lost when the function returns; it is saved for use during subsequent calls.

There are many uses of STATIC variables internal to functions. As an example, consider the function START_NEW_PAGE in the billing program. It is the only part of the program that refers to the page-number value, so the variable that holds the page number can be declared internal to it. But it won't do to use an AUTOMATIC variable. The most recent page number has to be saved after the function returns, so that on the next call to START_NEW_PAGE it can be stepped by 1 and printed out at the top of the next page. What's required is a STATIC variable:

```
START_NEW_PAGE:
   FUNCTION;
      VARIABLE PAGES IS INTEGER STATIC ::= 0;
      PAGES := PAGES + 1;
      PAGER ('ACCT #  OLD BAL.  PAYMENT  PAST DUE  INTEREST  '
         'PURCHASE  NEW BAL.', LPAD ('PAGE ', 60), PAGES);
      PRINT;
      LINE_COUNT := 1;
   END START_NEW_PAGE;
```

Storage for PAGES is allocated and assigned zero when execution of the program commences. But its value can only be used within START_NEW_PAGE, because its declaration is internal to that function. Each time START_NEW_PAGE is called, PAGES is stepped by 1; because it is STATIC, its storage is not freed -- and its new value is not lost -- when the function returns.

Constants may also be declared internal to a function with STATIC storage, but it often doesn't make much difference, since constants cannot be assigned new values by the function. Thus the only difference between

```
CONSTANT PI IS REAL ::= 0.314159E+01;
```

and

```
CONSTANT PI IS REAL STATIC ::= 0.314159E+01;
```

lies in when storage for PI is allocated, and when it is freed. Either way, it can only be used within the function it is internal to, and it can never have a value other than 0.314159E+01.

Function definitions internal to a function

So far, all the functions we have defined have been program-level functions. Like our program-level variables, their names have been known throughout the program -- so that they could be called from any point in the program.

However, just as some variables are used only by a particular function, some functions are called only by a particular function. That is, one can think of a function as having its own local functions, just like it has its own local variables. And just as local variables may be declared within the body of the function to which they belong, local functions may be defined within the body of the function that uses them.

We can give an example from the billing program. The function PROCESS_INPUT_NAME is not called by any program-level statement; it is called only from within PROCESS_NEXT_ACCOUNT. Thus the former function could be defined within the body of the latter. In outline, the result would be as follows:

```
PROCESS_NEXT_ACCOUNT:
   FUNCTION;
    . . .
   PROCESS_INPUT_NAME;
    . . .
   PROCESS_INPUT_NAME:
      FUNCTION;
       . . .
      END PROCESS_INPUT_NAME;

   END PROCESS_NEXT_ACCOUNT;
```

Defined in this way, a call to PROCESS_INPUT_NAME may only be made from within PROCESS_NEXT_ACCOUNT; it will not be recognized as a valid function name at program level or in other functions of the program.

How is it useful to define a function within another function? For one thing, it makes the structure of the program clearer, just as declaring variables within functions does. When we sketched out the program-level part of the billing program, we postulated program-level functions to perform the program's major tasks; but PROCESS_INPUT_NAME was not among these functions. Rather, PROCESS_INPUT_NAME was postulated to handle a "sub-task" within the task of PROCESS_NEXT_ACCOUNT. This relationship is made clearer by placing PROCESS_INPUT_NAME within PROCESS_NEXT_ACCOUNT.

There's no reason one has to stop at two levels of functions-within-functions. In a really complex program, sub-tasks can be broken into sub-sub-tasks, and those into sub-sub-sub-tasks, and so on. Consequently, functions can be usefully nested within other function to depths of three or four or more. In fact, CS-4 places no limit except convenience on the number of outer functions in which a function may be contained. You just have to remember the general rule: a function may not be called from outside the innermost function that contains it.

What about variables internal to nested functions? Are declarations at the beginning of PROCESS_NEXT_ACCOUNT known within PROCESS_INPUT_NAME? Are declarations within PROCESS_INPUT_NAME known throughout PROCESS_NEXT_ACCOUNT? The answers are yes and no, respectively. In general, a variable or constant can be used throughout the function where it is declared; but it cannot be used outside the function it is internal to. We will give more precise rules for this in the next chapter, where we discuss the more general topic of recognition of names.

Names restricted to BEGIN blocks

So far our discussion has dwelled entirely on how the use of a name can be restricted to a particular FUNCTION -- by writing the name's declaration (for variables or constants) or definition (for functions) within the FUNCTION body. This is the most narrow view of things, however. The same rules apply to the declaration or definition of a name within BEGIN blocks.

The most common use of BEGIN blocks in this way is to take advantage of AUTOMATIC storage. When a variable is declared within a BEGIN block, the default storage type is AUTOMATIC (just as it is for a variable declared in a FUNCTION definition). Storage for the variable is allocated each time the block is entered, and freed each time the block is exited. If an array or STRING is declared within a BEGIN block, it may be allocated with varying size.

As an example, we present yet another example of the center-of-gravity program:

```
VARIABLE CENTER IS REAL,
   N IS INTEGER;
REPEAT
   READ_LINE (N);
   IF N = 0 THEN RETURN;
   IF N < 0 THEN
      BEGIN;
         PRINT ('*****ERROR: NEGATIVE NUMBER OF WEIGHTS');
         RETURN;
      END;
   BEGIN;
      VARIABLE WEIGHT, DISTANCE ARE ARRAY(N) REAL;
      READ_LINE (WEIGHT, DISTANCE);
      IF SUM (WEIGHT) = 0 THEN
         BEGIN;
            PRINT ('*****ERROR: WEIGHT SUM = 0');
            RETURN;
         END;
      CENTER := SUM (WEIGHT * DISTANCE) / SUM (WEIGHT);
      PRINT (WEIGHT, DISTANCE, CENTER);
   END;
END;
```

Note the order of actions in each pass through the loop: first N is read in, then WEIGHT and DISTANCE are allocated with size N, then the two arrays are filled with input values and the usual calculations are

performed. Finally, storage for WEIGHT and DISTANCE is freed, when control reaches the END of the BEGIN block they were declared in. This program has a feature that none of our previous ones had: it imposes no limit on the number of input weights. Whatever the value of N, it allocates the arrays large enough to hold all the input data.

22.0
RECOGNITION OF NAMES

So far, we have demonstrated a number of advantages to structuring a program into FUNCTIONs and BEGIN blocks. Briefly, these include:

1) Programs can be written in a more logical way, by dividing the problems into tasks, the tasks into sub-tasks, and so on.

2) When these tasks are programmed separately, the logical structure of the program is clearer to someone reading it or to a programmer debugging it.

3) FUNCTIONs and blocks can take advantage of AUTOMATIC storage, especially in using arrays.

4) Parameters and declarations internal to FUNCTIONs and blocks can be exploited to keep those sections as independent of each other as possible, simplifying both writing and debugging of large programs.

There remains a problem, however, that we have not dealt with. When we added a function definition to the center-of-gravity program, for instance, we introduced new program-level variables and called them WT and DIS to distinguish them from the parameters named WEIGHT and DISTANCE. But we avoided saying if this is really necessary. Must there be two different sets of names? Or can one take the obvious step of defining WEIGHT and DISTANCE as both program-level and parameter variables?

We skirted a similar problem in the last chapter. Because the variable I was used in STORE_FREE_ACCT_NUMBERS independently of the rest of the billing program, we declared I internal to that function. But I is also used, independently of all other functions, in PROCESS_INPUT_NAME. Can I be defined internal to that function, too? Or must another name be used instead?

There are many questions of a similar nature. Can a variable be defined at program level, and also internal to a function? When can a name be used for a function in one part of a program, and for a variable in

another part? Can a BEGIN block have the same name as a FUNCTION?

These are not trivial questions. Even moderately large programs employ a great many names, so that a great deal of checking would be required to insure that no name were used for two or more independent purposes. To produce really large programs, independent blocks and functions must be written by different programmers -- and keeping names unique would become a horrendous job of management. Thus it is highly desirable to let a name be used independently to serve different purposes in different parts of one program.

Of course, some rules are needed. We have to say exactly when a name may be multiply defined (you already know, for instance, that only one definition of a name may be at program level). We also have to say exactly where each definition of a name is recognized.

Structuring into blocks and FUNCTIONs proves ideal for these purposes. Using the concepts of block and FUNCTION, this chapter presents a simple and straightforward set of rules for the recognition of names. The rules eliminate most worry over duplication of names in independent parts of a program, and they spell out what effect the position of a definition has on the name defined.

General definitions

Toward the end of the last chapter, we indicated that FUNCTION definitions may be nested in other FUNCTION definitions, or in BEGIN blocks. Of course, BEGIN blocks may also be nested in other BEGIN blocks or FUNCTION definitions. Thus in general, a program may contain an arbitrary number of these segments, nested in some arbitrary way. The aim of this chapter is to give rules that hold for any scheme of nesting. To do this in a clear and precise way, some general definitions are required.

The term block, as we have been using it, refers to a BEGIN block. (Actually, there are three types of blocks in CS-4; the other two will be explained in volume 2 of this Primer.) A FUNCTION in CS-4 is really one of the five kinds of procedures. (Two more kinds will be introduced in Chapter 24; the remaining ones will be explained in volume 2.) The rules given in the following paragraphs apply with equal force to all three kinds of blocks and all five kinds of procedures. In the rest of this chapter, for simplicity's sake, we'll use the term "block" rather informally to mean any kind of block (including BEGIN blocks) and any kind of procedure (including FUNCTIONs).

A block is said to contain some object (statement, another block, or whatever) if the object appears in the body of the block. We have occasion to speak of the innermost block containing an object: this is that block which contains the object, and which does not contain any other block containing the object.

A name, for present purposes, is a word defined in any of the following ways: by a declaration statement, by a parameter declaration in a FUNCTION heading, as a label to a BEGIN block, as the name of a CS-4 pre-defined function, or as the name of a FUNCTION which you define. (As with blocks, there are other types of names we have not yet encountered.) A definition of a name is internal to a block if:

1) the block is the innermost block that contains some declaration that defines the name; or

2) the block is a FUNCTION that defines the name as a parameter in the heading; or

3) the block is the innermost block that contains some other block, and the name is a label on this other block.

Each definition of a name is thus internal to at most one block. A definition not internal to any block is said to be external (or program-level, as we have been calling it). Names of program-level functions, and names declared in program-level declarations, are thus externally defined; so are the names of all pre-defined functions. A parameter name, on the other hand, is always internal to some FUNCTION.

A reference to a name is any use of the name apart from its definition. Thus a function call refers to a function name, an EXIT statement may refer to a BEGIN block name, an arithmetic expression may refer to any number of variable names, and so on.

Restrictions on the definition of names

A name may be defined any number of times within a program. But the placement of its definitions is limited by the following rules:

Rule 1: there may be only one external definition of a given name (in any one program);

Rule 2: only one definition of a given name may be internal to any one block.

The first rule yields the simpler restriction you already know: two program-level variables cannot have the same name. It also says that two program-level functions cannot have the same name, that a program-level function cannot have the same name as a program-level variable or BEGIN block, and so forth.

The second rule is just a generalization of the restrictions. It says that a name can be defined in only one way internal to each block.

Rules for references to names

What does it signify for a name to be defined more than once in a program? It means that the compiler will interpret a reference to the name in different ways, depending on where the reference occurs. A reference in one part of the program will be interpreted according to one definition, in other parts according to other definitions. Thus, it is not enough to just have rules that tell where a name may be defined legally. There must be a rule that tells where a name can be referred to legally. And there must be a rule that says which definition applies to a reference in a particular place.

In short, we need a way to determine precisely that area of a program where a given definition is applied when the name it defines is referred to. This area of application -- known as the scope of the definition -- is determined as follows:

Rule 3: a non-external definition of a name is applied throughout the body of the block it is internal to -- except in the bodies of any contained blocks that have a different definition of the name internal to them. In other words, the scope of the definition is the body of the block it is internal to, minus any scopes for the same name that are contained in that block.

Rule 4: an external definition of a name is applied throughout the program -- except in the bodies of any blocks with different definitions of the name internal to them.

Thus the definition's scope is the whole program, minus any other scopes for the same name that are contained in the program.

Rules 1 through 4 together guarantee that no two scopes for definitions of the same name ever overlap in a valid program. So there is no question which definition of a name the compiler applies at any given point. Of course, there may be some places where a name is undefined; these cases are governed by one last rule:

Rule 5: a name may only be referred to within the scope of one of its definitions; otherwise, the reference is in error.

Each of the rules of this section is a general case of one you already know. Rule 4 implies that a name defined only at program level is known throughout the program. Rule 3 implies that a name defined only as a parameter or internal name within one function is known throughout that function; and rule 5, such a name is not known outside the body of the one function where it is defined.

An example with multiple declarations

The rules we have given are best appreciated by seeing how they work together in actual examples. We, therefore, present here the entire billing program of Chapter 16, incorporating many of the changes introduced by subsequent chapters, plus a few brand new changes. In particular, there is now one name that appears in three different declarations, and another that appears in two:

```
VARIABLES I, PAST_COUNT, ACCT_NO ARE INTEGER ::= 0,
   LINE_COUNT IS INTEGER ::= 50;
VARIABLE FREE_ACCT_TABLE IS ARRAY(999) BOOLEAN ::= FALSE;

STORE_FREE_ACCT_NUMBERS;
FOR I UNTIL ACCT_NO <= 0
   REPEAT
      LINE_COUNT := LINE_COUNT + 1;
      IF LINE_COUNT = 51 THEN START_NEW_PAGE;
      PROCESS_NEXT_ACCOUNT;
   END;
PRINT_STATISTICS;

STORE_FREE_ACCT_NUMBERS:
   FUNCTION;
      VARIABLES I, N, FREE_ACCT ARE INTEGER;
      READ_LINE (N);
      IF N < 0 | N > 999 THEN
         TERMINATE_WITH_ERROR ('NUMBER OF FREE ACCOUNTS ' ||
            'NEGATIVE OR >999');
      FOR I THRU N
         REPEAT
            READ_LINE (FREE_ACCT);
            IF FREE_ACCT < 1 | FREE_ACCT > 999 THEN
               TERMINATE_WITH_ERROR ('FREE ACCOUNT NUMBER IS '
               || '<1 OR >999');
            FREE_ACCT_TABLE(FREE_ACCT) := TRUE;
         END;
   END STORE_FREE_ACCT_NUMBERS;
```

```
START_NEW_PAGE:
   FUNCTION;
      VARIABLE PAGES IS INTEGER STATIC ::= 0;
      PAGES := PAGES + 1;
      PAGER ('ACCT #   CUSTOMER NAME    OLD BAL.  ',
         'PAYMENT  PAST DUE  INTEREST  PURCHASE  ',
         'NEW BAL.', LPAD ('PAGE ', 60), PAGES);
      PRINT;
      LINE_COUNT := 1;
   END START_NEW_PAGE;

PROCESS_NEXT_ACCOUNT:
   FUNCTION;
      VARIABLES OLD_BAL, PAYMENT, PURCHASE,
         NEW_BAL ARE REAL,
         PAST_DUE, INTEREST ARE REAL ::= 0;
      VARIABLE NAME IS STRING(20);
      CONSTANT ERROR_MESSAGE IS STRING(23) ::=
         'ACCOUNT HAS NUMBER >999';
      READ_LINE (NAME, ACCT_NO, OLD_BAL, PAYMENT, PURCHASE);
      NAME := NAME_CHECK (NAME);
      IF ACCT_NO <= 0 THEN RETURN;
      IF ACCT_NO > 999 THEN
         TERMINATE_WITH_ERROR (ERROR_MESSAGE);
      IF OLD_BAL = 0 & PAYMENT = 0 & PURCHASE = 0 THEN
         BEGIN;
            PRINT ('  ', LPAD (ACCT_NO, 3, '0'), ' ', NAME);
            RETURN;
         END;
      IF ~FREE_ACCT_TABLE(ACCT_NO) & PAYMENT < OLD_BAL THEN
         BEGIN;
            PAST_DUE := OLD_BAL - PAYMENT;
            INTEREST := PAST_DUE * 0.015;
            PAST_COUNT := PAST_COUNT + 1;
         END;
      NEW_BAL := OLD_BAL - PAYMENT + INTEREST + PURCHASE;
      PRINT ('  ', LPAD (ACCT_NO, 3, '0'), ' ', NAME,
         DSTRING (OLD_BAL, 5, 2), DSTRING (PAYMENT, 5, 2),
         DSTRING (PAST_DUE, 7, 2), DSTRING (INTEREST, 7, 2),
         DSTRING (PURCHASE, 7, 2), DSTRING (NEW_BAL, 7, 2));
```

```
   NAME_CHECK:
      FUNCTION (INPUT IS STRING(20)) STRING(*);
         VARIABLES I, FIRST, LAST ARE INTEGER;
         VARIABLE ERROR_MESSAGE IS STRING(31) ::=
            'INVALID CHARACTER WITHIN A NAME';
         FOR FIRST THRU 20 UNTIL INPUT(FIRST)~= ' '
            REPEAT; END;
         IF INPUT(FIRST) = ' ' THEN RETURN;
         FOR LAST FROM 20 BY -1 UNTIL INPUT(LAST) ~= ' '
            REPEAT; END;
         FOR I FROM FIRST THRU LAST REPEAT
            IF (INPUT(I) < 'A' | INPUT(I) > 'Z') &
               INPUT(I) ~= ' ' & INPUT(I) ~= '''' THEN
               TERMINATE_WITH_ERROR (ERROR MESSAGE);
            END;
         RETURN INPUT(LAST - FIRST + 1 AT FIRST);
      END NAME_CHECK;

   END PROCESS_NEXT_ACCOUNT;

PRINT_STATISTICS:
   FUNCTION;
      PRINT; PRINT;
      PRINT (I || ' ACCOUNTS WERE PROCESSED');
      PRINT (PAST_COUNT || ' ACCOUNTS HAD BALANCES PAST DUE');
   END PRINT_STATISTICS;

TERMINATE_WITH_ERROR:
   FUNCTION (MESSAGE_OUTPUT IS STRING(*));
      IF I = 0 THEN PRINT ('EXECUTION TERMINATED BY ' ||
         'ERROR BEFORE PROCESSING: ', MESSAGE_OUTPUT);
      ELSE PRINT ('EXECUTION TERMINATED BY ' ||
         'ERROR DURING PROCESSING: ', MESSAGE_OUTPUT);
      TERMINATE;
   END TERMINATE_WITH_ERROR;
```

ERROR_MESSAGE is declared as a STRING(23) constant internal to PROCESS_NEXT ACCOUNT. It is also declared a STRING(31) variable internal to NAME_CHECK, a function which is contained in PROCESS_NEXT_ACCOUNT. Thus the scope of the constant ERROR_MESSAGE is all of the body of PROCESS_NEXT_ACCOUNT, except for the body of NAME_CHECK; the body of NAME_CHECK is the scope of the variable ERROR_MESSAGE. This means that the compiler interprets the statement

```
TERMINATE_WITH_ERROR (ERROR_MESSAGE);
```

as referring to a variable STRING(31) when it appears within NAME_CHECK's body, and to a constant STRING(23) when it appears within PROCESS_NEXT_ACCOUNT but not NAME_CHECK.

Notice that only the ability to refer to the STRING(23) data is lost when NAME_CHECK is called -- the storage area for the data remains allocated, even though a second, STRING(31) storage area for ERROR_MESSAGE is also allocated. After NAME_CHECK returns, the STRING(23) may again be referred to with the name NAME_CHECK, in the rest of PROCESS_NEXT_ACCOUNT. This illustrates an important point. Scopes are not determined by relations between data types or by the flow of control; they depend only on position within the program. Several pieces of data with the same name may be in storage at the same time -- but only one of them can be referred to in any one scope.

We can demonstrate this point again by considering the variable I. There are two AUTOMATIC variables called I -- one whose scope is the body of NAME_CHECK, and one whose scope is the body of STORE_FREE_ACCT_NUMBERS. There is also an external STATIC declaration of I, whose scope is the rest of the program. Storage for this latter I is maintained throughout the program's execution -- but its value cannot be referred to within the body of NAME_CHECK or STORE_FREE_ACCT_NUMBERS. Now consider what happens when NAME_CHECK is called with an invalid name. The I used as a FOR variable is NAME_CHECK's internal one. But when TERMINATE_WITH_ERROR is called, control moves to the scope of the external I; thus the value that is tested in TERMINATE_WITH_ERROR is the value of the STATIC I, which is what we want to test. Storage for NAME_CHECK's I is still allocated when TERMINATE_WITH_ERROR is being executed, but the value being stored can no longer be referred to -- since the program is terminated without control returning to NAME_CHECK's body.

In our examples, a name declared several times still represented the same type of data in each declaration. This is most often the case, because names are usually chosen for their mnemonic significance -- NAME is more likely to be a STRING than a REAL, ACCT_NO is more likely an INTEGER than a BOOLEAN. But none of this is required by the rules. A name may refer to data of any number of different modes in different scopes.

Scopes of parameter names

Under the scope rules, a function call may contain argument variables whose names are the same as their corresponding parameters. This is because the parameters' scope lies entirely within the body of the function that declares them. The call is outside the function's body, where the arguments are interpreted according to declarations with a different scope. (Some special rules apply when a function is called from within its own body. But that is an advanced topic -- all of the functions we have showed how to define thus far may not legally be called from within themselves.)

As an example, we can now write the center-of-gravity program (p. 209) with external arrays named WEIGHT and DISTANCE instead of WT and DIS. The scope of the parameters with these names is limited to the body of GRAVITY_CENTER; the scope of the external declarations is the rest of the program, including the call to GRAVITY_CENTER.

Names used in parameter declarations

Some parameter declarations use names other than the ones they declare. For instance, variables or constants may be used to specify array parameter sizes, as you saw in Chapter 17. Each use of a variable or constant in a parameter declaration constitutes a reference, which is interpreted according to the name scope rules.

The interpretation of most such references is obvious. But there is one tricky case. Consider a variable used in a parameter declaration, and declared internal to the function as well:

```
VARIABLE N IS INTEGER;
. . .
CENTER:
   FUNCTION (WEIGHT, DISTANCE ARE ARRAY(N) REAL);
      VARIABLE N IS REAL;
      . . .
```

The scope of the REAL N is the body of the function CENTER, while the scope of the INTEGER N is the text external to the definition of CENTER. However, the N in the array specification (which is part of the heading), refers to the INTEGER N. Any use of N in the body of the function, though, will be a reference to the REAL N.

Names used in declaration statements

Variable or constant names are used in declarations internal to a function the same way they are used in parameter declarations -- to specify array sizes. They are also used in initialization expressions. If the names used are only declared outside of the function, then the scope rules can be applied without raising any problems. However, it is a different matter when a name used in a declaration was itself declared internal to the same function.

To see why, consider first this example:

```
ABC1:
   FUNCTION;
      VARIABLE A IS REAL ::= B;
      VARIABLE B IS REAL ::= A;
      . . .
```

The scope rules say that the B and the A in the initialization expressions refer to the B and A internal to ABC1. Thus each variable is being initialized to the other -- clearly a circular definition. Now consider this possibility:

```
ABC2:
   FUNCTION;
      VARIABLE A IS REAL ::= 1;
      VARIABLE B IS REAL ::= A;
      . . .
```

This seems as though it should work. First A is initialized to 1, then B is initialized to A. But what if the declarations were reversed:

```
ABC3:
   FUNCTION;
      VARIABLE B IS REAL ::= A;
      VARIABLE A IS REAL ::= 1;
      . . .
```

Is this an error, because B is initialized to A before A has any initial value? Or does the compiler recognize this, and initialize A first, even through its declaration comes second?

CS-4 resolves such cases by a simple method. Expressions used in declarations of variables and constants (such as expressions for initialization or array-size values) are always evaluated in the order that the declarations appear in the function body. If a name is used in a declaration that precedes its own declaration internal to the same block, it is in error. Thus function ABC2 is correct, while ABC1 and ABC3 are wrong.

Since parameters are always the first names declared internal to a function, they may be used legally anywhere in subsequent internal declarations. And, of course, names declared externally or in containing functions, are not subject to the restrictions we have described.

In external declarations, only external names can be used -- they are the only ones at program level. The same rule applies: names cannot be used in external declarations that come before their own.

Scope of function names

The scope rules affect functions much as they do variables and constants. A name may be defined to refer to different functions in different parts of a program. One definition may be superceded by another with the same name in a contained block.

Both declarations and function definitions create new scopes for a name. Thus a name cannot be defined as both a variable and a function internal to the same block. Furthermore, a variable or constant declaration overrides any function definition in a containing block (or at program level), and vice versa. This can have serious consequences: if PRINT is declared as an INTEGER, you cannot refer to the PRINT function within the scope of that declaration.

It is important to realize that a function definition can only be internal to a block that contains it -- it is never in any sense internal to itself. Thus it is perfectly legal to write

```
TEST1:
   FUNCTION;
   . . .
   TEST2:
      FUNCTION;
         VARIABLE TEST2 IS BOOLEAN;
         . . .
      END TEST2;

   END TEST1;
```

The BOOLEAN TEST2 is internal to the FUNCTION definition TEST2, while the function TEST2 is internal to the FUNCTION definition TEST1. Thus the scope of the TEST2 variable name is the body of TEST2, while the scope of the TEST2 function name is all of the body of TEST1 outside the body of TEST2.

Function names, like variable or constant names, may be used in declarations. For example:

```
SAMPLE:
   FUNCTION (X IS ARRAY (SZ (Y)) REAL);
      VARIABLE USER IS STRING (LTH (NAME)) ::= ' ';
      CONSTANT N IS INTEGER ::= SUM (X);
      . . .
```

There is a difficulty that may arise here in one particular case: if the function LTH is itself defined internal to SAMPLE. If LTH happens to refer to the variables USER or N which are being declared, it must be an error -- neither has been initialized yet, and the length of USER is not even determined. It is not possible for the compiler to check for all errors of this sort. Instead, the language imposes a stronger restriction: a declaration statement may not call a function internal to the same block. For the same reasons, external declarations may not call any functions defined in the program; they may call pre-defined functions, however.

Scopes of BEGIN block names

The scope rules make it possible to use the name of a BEGIN label for unrelated purposes elsewhere in a program, without committing an error. In addition, it is occasionally desirable to have two or more BEGIN blocks with the same name. In such a case, the scope rules are used to determine which block is referred to by a FROM phrase of an EXIT statement. In particular, if a statement such as

```
EXIT FROM TEST;
```

is within two or more BEGIN blocks named TEST, only the innermost one will be terminated -- because it is the scope of the innermost definition of TEST that contains the EXIT statement.

23.0

MORE PRE-DEFINED FUNCTIONS

CS-4 provides many functions for your use, in addition to the ones we have already described. This chapter provides brief summaries of many of them; a complete listing can be found in the Language Reference Manual.

Trigonometric functions

Pre-defined functions are available to compute both the natural trigonometric and the hyperbolic functions, and their inverses. Each accepts a REAL (or convertible to REAL) argument, and returns a REAL:

Function	Value computed	
SIN	sine	of argument in radians
COS	cosine	
TAN	tangent	
COT	cotangent	
SEC	secant	
CSC	cosecant	
ARCSIN	inverse of sine	result in radians
ARCCOS	inverse of cosine	
ARCTAN	inverse of tangent	
SINH	hyperbolic sine	
COSH	hyperbolic cosine	
TANH	hyperbolic tangent	
COTH	hyperbolic cotangent	
SECH	hyperbolic secant	
CSCH	hyperbolic cosecant	
ARCSINH	inverse of hyperbolic sine	
ARCCOSH	inverse of hyperbolic cosine	
ARCTANH	inverse of hyperbolic tangent	

An error is signalled if any of these functions is called with a value outside of its domain, or a value that would produce an infinite result (details are in the table for trigonometric functions in the Language Reference Manual).

Logarithms and exponentials

There are four pre-defined functions in this category. Each produces a REAL result with a single REAL (or convertible to REAL) argument:

Function	Value computed
LOG	logarithm to the base 10
LN	natural logarithm (to the base e)
LOG2	logarithm to the base 2
EXP	e to the power of the argument (uses a faster algorithm than the ** operator)

An error is signalled if one of the three logarithmic functions is called with a non-positive argument.

Other functions on single numerical values

Several other pre-defined functions on INTEGER or REAL values are provided because of their convenience in certain applications. Each takes a single argument.

SQRT. Computes the square root of the argument. SQRT (X) employs a more efficient algorithm than X ** 0.5. The argument must be REAL or convertible to REAL, and the result is REAL.

ABS and SGN. ABS returns the absolute value of the argument. SGN returns 1 if the argument is positive, -1 if it is negative, and 0 if it is zero. (Thus, for any X,

```
X = ABS (X) * SGN (X)
```

is TRUE.) ABS returns a REAL when called with a REAL argument, and an INTEGER when called with an INTEGER argument; SGN returns an INTEGER, with either a REAL or INTEGER argument.

CEIL, FLOOR and TRUNC. CEIL (short for ceiling) returns the smallest integral value not less than the argument; FLOOR returns the largest integral value not greater than the argument. TRUNC returns the

argument value with its fractional part truncated. Thus TRUNC (X) = CEIL (X) for X ≤ 0, and TRUNC (X) = FLOOR (X) for X ≥ 0. All three functions take a REAL argument, and return an INTEGER.

Array handling

A number of pre-defined functions are provided to make it easier to work with arrays. You have seen most of these already, but we repeat them here for convenience.

SIZE. Returns an INTEGER which is the size of one dimension of an array. SIZE (A), where A is an array of any element mode, finds the size of the first dimension of A. SIZE (A, N), where N is an INTEGER, finds the size of the Nth dimension of A.

ALL and ANY. The ALL function applies & between all the elements of its argument; ANY applies | between the elements. Both accept a BOOLEAN array of any rank, and return a BOOLEAN.

SUM and PRODUCT. The SUM function computes the sum of its argument's elements; PRODUCT computes their product. Both return a REAL when called with an ARRAY(n) REAL, and an INTEGER when called with an ARRAY(n) INTEGER (where n represents an arbitrary size value).

MAX and MIN. MAX returns the greatest element of its argument, and MIN returns the least element. Both produce an INTEGER value for a rank 1 INTEGER array, and a REAL value for a rank 1 REAL array.

NDXMAX and NDXMIN. For an INTEGER or REAL array A of one dimension, NDXMAX (A) is the least index of MAX (A) -- in other words, NDXMAX (A) is equal to the lowest N such that A(N) = MAX (A). NDXMAX thus tells where the first occurrence of the maximum element in A appears. Analogously, NDXMIN (A) is the least index of MIN (A). Both functions return a single INTEGER value.

String handling

The following pre-defined functions are provided for working with character strings. In every case, where an argument of STRING mode is indicated one may substitute an argument of a mode convertible to STRING (such as REAL, INTEGER, or BOOLEAN).

LENGTH. Takes a STRING-mode argument, and returns an INTEGER which is the length of the string. (LENGTH is equivalent to SIZE, except that it can only be applied to STRING values; it is included for mnemonic convenience.)

SUBSTR. This function selects a substring from a string. It thus has the same effect as string subscripting.

SUBSTR may be called with two or three arguments; the first is STRING, the others INTEGER. SUBSTR (S, N, L) returns a substring of S of length L, beginning with the Nth character of S; in other words, it returns S(L AT N) if L is positive, and the empty string if L is zero. The substring SUBSTR (S, N) contains all characters of S from the Nth to the last; hence it is the same as S(LENGTH (S) - N + 1 AT N)). A call to SUBSTR is in error if N is non-positive or greater than LENGTH (S); or if L is negative or greater than LENGTH (S) - N + 1.

INDEX AND RINDEX. These functions search a string for the occurrence of particular substring. Each takes two STRING arguments. INDEX (S, SUB) searches for the first occurrence of SUB as a substring of S, while RINDEX (S, SUB) searches for the last occurrence of SUB in S (in other words, it searches from right to left). Both return an INTEGER that indicates at which character of S the found occurrence of SUB begins; if SUB is not found in S, they return 0. Here are some examples:

Expression	Value
INDEX ('ABCDE', 'BC')	2
RINDEX ('ABCDE', 'BC')	2
INDEX ('ABCABC', 'BC')	2
RINDEX ('ABCABC', 'BC')	5
INDEX ('ABCABC', 'Z')	0

A call to INDEX or RINDEX is in error if the length of the second argument exceeds the length of the first.

PAD. This function "right pads" a string with spaces. Its first argument is a STRING, and its second an INTEGER. PAD (S, N) returns a STRING value of length N, whose first LENGTH (S) characters are the characters of S, and whose remaining characters are blank characters. A call to PAD is in error if N is less than the length of S.

Operating system interface functions

PRINT, OVER_PRINT, PAGER, READ_LINE, and TERMINATE perform their actions by communicating with the computer's operating system -- a program or group of programs that regulate how and when your program uses the different parts of the computer. These functions are thus known as operating system interface functions. PRINT, OVER_PRINT, PAGER, and READ_LINE request to use input or output devices connected to the computer; TERMINATE requests that a program be terminated.

There are many other operating system interface functions, all of which are described in the Operating System Interface manual. We will not explore these functions further in this Primer, however, since generally they require a more than elementary knowledge of the properties of operating systems.

24.0
DEFINITION OF OPERATORS

Operators in CS-4 work in very much the same way that functions do. Consider these similarities:

1) Each operator, like each function, has a distinct name that distinguishes it from other operators or functions.

2) Each reference to an operator name specifies some particular set of actions -- we have been calling it an operation -- that is to be performed; this is analogous to what a function does.

3) The actions performed by an operator depend on its arguments -- or operands, as we have called them.

4) An operator may have a return value, which we have been referring to as a result value.

There are indeed, only two important differences between functions and operators. First, there are different rules for forming their names -- operators may be composed of certain special symbols. Second, the syntax of their calls differs. A prefix operator is placed directly before its single argument; an infix operator is placed between its two arguments. Operator calls are thus limited to one or two arguments, while functions may have any number of arguments, even none.

We mentioned earlier that CS-4 had other "procedures" in addition to FUNCTIONs. As you may have guessed by now, prefix operators and infix operators are two of those procedures. These operators, like FUNCTIONs, may be defined by the programmer. This chapter will show how this is done. As you will see, they are defined in almost exactly the same way that functions are defined.

Operator names

All the operators we have used so far are named by operator symbols. Programmer-defined operator symbols may be composed of one or more of the following characters:

```
! $ % & * + - / : < = > ? @ \ ^ | ` ~ .
```

subject to three restrictions:

1) An operator symbol may not begin or end with the decimal-point character(.).

2) The operator symbol may not exceed 32 characters in length.

3) Four operators are reserved:

```
.  :  :::  ::=
```

Like the reserved words discussed in Chapter 2, these have fixed meanings in CS-4, and may not be redefined in a program.

The following are thus examples of valid operator symbols:

```
==  +:+  +.*  ?
```

while these are all illegal names for operators:

```
.+  :  //.  (&)  +,*  7=  PLUS@  ...
```

Operators may also be named by identifiers, just as variables and functions are. We have shown no examples of this so far, but we can mention here that the logical operators &, |, and ~ may also be referred to with the names AND, OR, and NOT, respectively. Thus the expression

```
~A & ~B | C
```

may equivalently be written as

```
NOT A AND NOT B OR C
```

Note that two adjacent identifiers, like adjacent operators, must be separated by a space so that they can be distinguished by the compiler.

Defining prefix and infix operators

A prefix operator is defined by writing a PREFIXOP definition. It is identical to a FUNCTION definition, except that the word FUNCTION is replaced by PREFIXOP. Also, a PREFIXOP definition must declare exactly one parameter, or it is in error.

An infix operator is defined by writing an INFIXOP definition. It, too, is identical to a FUNCTION definition, except that the word INFIXOP replaces FUNCTION. An INFIXOP must declare exactly two parameters. The first parameter is passed the left operand, and the second is passed the right operand, unless you specify otherwise (we will show how to do so later in this chapter).

The rules we have given that relate to FUNCTIONs also apply to operators. Thus the label (or labels) on the heading of a PREFIXOP or INFIXOP definition name the operator that is being defined; these labels may be either operator symbols or identifiers. Parameters and return values are manipulated in the ways you already know, and the mechanics of call and return are the same.

PREFIXOP and INFIXOP definitions obey the same rules as all procedures and blocks with respect to storage allocation and the scope of names. There is an important extension of the scope rules, however, when they apply to operator names. Essentially, an operator may be defined as both an infix and a prefix operator at the same time. More exactly we can say:

1) A name may be defined as both a prefix operator and an infix operator internal to the same block or procedure.

2) The scope of a prefix operator's name is the body of the block or procedure to which its definition is internal, except for the bodies of any contained blocks or procedures internal to which the name is redefined as something other than an infix operator. And vice-versa -- this same rule holds if you switch the words infix and prefix.

You have already seen two pre-defined operators, + and -, which are both infix and prefix. This causes no problems of interpretation, because the context of an operator always determines whether it is intended as infix or prefix.

A procedure invocation is anything that causes a procedure's body to be executed; a function call is thus an invocation, as is the operation of an operator on one or two operands.

One further fact about procedures is useful here. The heading of a procedure may be labelled with two or more names, in which case any one of the names may be used to refer to the function or operator being defined. For example:

```
& : AND:
   INFIXOP (A, B ARE BOOLEAN) BOOLEAN;
```

might be the heading for the logical operator that can be invoked by either & or AND.

Examples of operator definitions

Our first example is a prefix operator which sums over the first dimension of a two-dimensional array of REALs, like the function SUM_2 of chapter 19:

```
+/ : SUM2:
   PREFIXOP (ARRAY_2 IS ARRAY(*,*) REAL) ARRAY(*) REAL;
      VARIABLE I IS INTEGER,
         ARRAY_SUM IS ARRAY(*) REAL ::= ARRAY_2(1);
      FOR I FROM 2 THRU SIZE (ARRAY_2, 1) REPEAT
         ARRAY_SUM := ARRAY_SUM + ARRAY_2(I); END;
      RETURN ARRAY_SUM;
   END +/;
```

If DISTANCE is declared as

```
VARIABLE DISTANCE IS ARRAY(M,N) REAL;
```

then either of the following:

```
+/DISTANCE
SUM2 DISTANCE
```

represents an ARRAY(N) REAL equal to the sum over the first dimension of DISTANCE. Notice that, since +/ and : are both operator symbols, they must be separated by a space in the definition.

We will give two examples of infix operator definitions. The first one raises an integral value (left operand) to a non-negative integral power (right operand). It takes two INTEGERs and returns an INTEGER:

```
*** :
   INFIXOP (BASE, EXPONENT ARE INTEGER) INTEGER;
      VARIABLE PARTIAL_PRODUCT IS INTEGER ::= 1;
      IF EXPONENT < 0 THEN
         BEGIN;
            PRINT ('*****ERROR: NEGATIVE'
               || ' EXPONENT TO *** OPERATOR');
            PRINT ('     PROGRAM TERMINATED BY ERROR');
            TERMINATE;
         END;
      THRU EXPONENT REPEAT
         PARTIAL_PRODUCT := PARTIAL_PRODUCT * BASE; END;
      RETURN PARTIAL_PRODUCT;
   END ***;
```

This operator can be invoked, in the scope where it is known, by expressions such as:

```
A *** B
2 *** SIZE_LIMIT
X *** 0
Y *** -3
```

If SIZE_LIMIT above is a REAL, its value is converted to INTEGER by rounding before it is assigned to the parameter EXPONENT. The invocation

with right operand -3 causes the procedure to terminate the program, as would either of the first two invocations if B or SIZE_LIMIT were negative.

Our second infix operator takes a STRING value as right operand, and a STRING variable as left operand, and assigns the former to the latter. If the value to be assigned is longer than the variable, excess characters at the right of it are discarded (in other words, the right operand's value is truncated at the right, if necessary).

```
|= :
   INFIXOP (TARGET IS STRING(*) OUTPUT, VALUE IS STRING(*));
      IF LENGTH (TARGET) >= LENGTH (VALUE) THEN
         TARGET := VALUE;
      ELSE TARGET := VALUE (LENGTH (TARGET) AT 1);
   END |=;
```

This is an example of a useful operator that does not need a return value. It passes its result back to the left operand through an OUTPUT parameter. (Of course, you could give it a return value, too, if you wished.)

Precedence in operator definitions

When a programmer-defined operator is used with other operators in an expression, what is its relative precedence? When no precedence is specified in the definition -- as in those of the previous section -- the defined operator is given a default precedence which is just greater that that of := (but which is less than the precedence of any previously defined operator whose precedence is greater than :=).

It is sometimes valuable, though, to be able to define an operator with some other precedence relative to previously defined operators. This is done within an ATTRIBUTES phrase which is placed at the end of the INFIXOP or PREFIXOP heading. Here is an example of an ATTRIBUTEs phrase that specifies the precedence of *** :

```
*** :
   INFIXOP (BASE, EXPONENT ARE INTEGER) INTEGER
      ATTRIBUTES (PRECEDENCE = **);
```

This says that *** has the same precedence as **, a pre-defined operator.

The expression

```
PRECEDENCE = oper
```

where "oper" is the name of some operator, is called an attribute. There are two other forms of attributes that may be used to specify precedence:

```
PRECEDENCE > oper
PRECEDENCE < oper
```

The first says that the operator being defined has a precedence just greater than the one named "oper"; in other words, the defined operator has greater precedence than "oper", but less precedence than any previously defined operator with precedence greater than "oper". Similarly, the other attribute above says that the defined operator has precedence just less than "oper", in the same sense.

The presence of some "oper" in a precedence attribute is considered a reference to that name in the definition heading. Thus the attribute is in error if it is not within the scope of some definition of "oper". If "oper" is defined as both infix and prefix operator in the scope that contains the precedence attribute, it is assumed to refer to the infix one if the attribute is in an INFIXOP definition heading, and to the prefix one if it is in a PREFIXOP definition.

The names =, >, <, and PRECEDENCE, when they appear in a precedence attribute, are not interpreted as references to definitions of these symbols. Instead, they have special context meanings, which are independent of where or whether they have been defined in the program. Thus it is even possible to write

```
ATTRIBUTES (PRECEDENCE > >)
```

which specifies a precedence just greater than that of >. The first > symbol is interpreted with the special context meaning; the second > symbol refers to a defined operator > whose scope contains the procedure heading in which the ATTRIBUTES phrase appears.

(We have encountered names with special context meanings before, although we did not identify them as such. Among them are the storage type names STATIC and AUTOMATIC, the parameter type names INPUT, OUTPUT, INOUT, and COPYIN, and the operator symbol * used as an abbreviation for a subarray subscript. The names used in the attributes defined in the next two sections are also interpreted with special context meanings.)

Associativity in operator definitions

Precedence is just one of the two factors that determine the grouping of operands. The other is associativity. The associativity of a defined infix operator may be specified explicitly with one of the following attributes:

```
LEFT ASSOCIATIVE
RIGHT ASSOCIATIVE
```

Each attribute is written as two words.

The default associativity attribute is LEFT ASSOCIATIVE. So only RIGHT ASSOCIATIVE need be written explicitly in an ATTRIBUTES phrase:

```
*** :
   INFIXOP (BASE, EXPONENT ARE INTEGER) INTEGER
      ATTRIBUTES (RIGHT ASSOCIATIVE);
```

How does one specify both an associativity and a precedence attribute? When a definition contains two or more attributes, they are placed in a comma-separated list in the ATTRIBUTES phrase:

```
*** :
   INFIXOP (BASE, EXPONENT ARE INTEGER) INTEGER
      ATTRIBUTES (RIGHT ASSOCIATIVE, PRECEDENCE = **);
```

The order of the attributes does not matter. There may not be more than one ATTRIBUTES phrase in a definition, nor may any type of attribute appear more than once in the ATTRIBUTES phrase.

The COMMUTATIVE attribute

It is sometimes convenient to let the arguments to an infix operator appear in either order -- that is, to let the left operand be assigned to the second parameter, and the right operand to the first parameter, instead of the other way around. This property can be indicated by adding an attribute consisting of the single word

```
COMMUTATIVE
```

To see how COMMUTATIVE works, consider the following operator which returns a string concatenated with itself an integral number of times:

```
|*| :
   INFIXOP (UNIT IS STRING(*), N IS INTEGER) STRING(*)
      ATTRIBUTES (COMMUTATIVE);
      IF N <= 0 THEN
         BEGIN;
            PRINT ('*****ERROR: |*| INVOKED WITH NON-' ||
               'POSITIVE ARGUMENT.');
            TERMINATE;
         END;
      BEGIN;
         CONSTANT L IS INTEGER ::= LENGTH (UNIT);
         VARIABLE OUTSTRING IS STRING(N * L) ::= UNIT,
            I IS INTEGER;
         FOR I FROM L THRU (N - 1) * L BY L REPEAT
            OUTSTRING(L AT I + 1) := UNIT; END;
         RETURN OUTSTRING;
      END;
   END |*|;
```

This operator may be invoked with either an INTEGER left operand and STRING right operand, or a STRING left operand and INTEGER right operand. Either way, the STRING is concatenated with itself as many times as the INTEGER specifies. For instance,

```
3 |*| 'ABC'
'ABC' |*| 3
```

both return the value 'ABCABCABC'.

The situation is a bit more complicated when the operand modes do not match the parameter modes, even when the order of the operands is reversed. The compiler then tries to find a suitable conversion for one or both operands. It first looks for a conversion of the left operand to the first parameter's mode, and the right operand to the second parameter's mode; if this doesn't work, it tries to find conversion for the reverse order. Here are some examples, using the |*| operator:

Expression	Value
'ABC' |*| 3.7	'ABCABCABCABC'
3.7 |*| 'ABC'	'ABCABCABCABC'
3 |*| 2	'33'
2 |*| 3	'222'
2 |*| 2.9	'222'

The order of the operands is reversed in the second example -- because there is no conversion from the STRING 'ABC' to INTEGER. In all the other cases, conversions can be found for the operands in the order they appear.

The assignment operator

Now that we have explained the concepts which underlie operator definitions, we present some additional information about how the assignment operator := has been defined and how it operates.

The assignment operator is defined for every mode used in a program's declarations, including modes that are used as part of generated modes (that is, if you declare an ARRAY(10) BOOLEAN), := is defined both for ARRAY(10) BOOLEAN and for BOOLEAN). When the left operand is of a mode generated with ARRAY, and the right operand is of the array's element-mode, := is applied according to the rules we have given for array assignment.

The return value of := is the value of the left operand after the assignment has been performed. One may thus assign the same value

to more than one variable with a statement such as:

```
FACT := (MULT := 1);
```

Assignment is right associative, so the parentheses above may be dispensed with. As another example, one could replace these two statements which we showed earlier in a prime-generating program:

```
TEMP := CANDIDATE / PRIME(I);
IF PRIME(I) * TEMP = CANDIDATE THEN EXIT FROM TEST;
```

with a single statement:

```
IF (PRIME(I) * (TEMP := CANDIDATE /PRIME(I))) = CANDIDATE
   THEN EXIT FROM TEST;
```

PRIME(I) is here multiplied by the value assigned to TEMP, which is the value of CANDIDATE / PRIME(I) after rounding.

Pre-defined operators

You already know most of the CS-4 pre-defined operators. They are the ones for arithmetic, comparison, and logical operations, and for concatenation. We mention here only one more, which has not come up in previous discussions. (A tabulation of all pre-defined operators used in this volume, along with their associativity and relative precedence, appears in Appendix B.)

Integer divide. The infix operator IDIV takes two INTEGER operands and returns an INTEGER value. A IDIV B is equal to A / B with its fractional part truncated; for example:

Value of A	Value of B	A / B	A IDIV B
5	4	1.25	1
15	4	3.75	3
-5	4	-1.25	-1
-15	4	-3.75	-3
10	5	2.	2

IDIV can be useful in programs concerned with the divisibility of one number by another. In a program that generates prime numbers, for instance, one could make use of this expression:

```
CANDIDATE = PRIME(I) * (CANDIDATE IDIV PRIME(I))
```

which evaluates to TRUE only if PRIME(I) evenly divides CANDIDATE.

APPENDICES

APPENDIX A

ASCII CHARACTER SET AND COLLATING SEQUENCE

Below is the full ASCII character set, including control characters. The character code determines the collating sequence -- characters with numerically lower codes precede those with higher codes.

Character Code	Graphic (if any)	Remarks
000		null
001		start of heading; start of message
002		start of text; end of address
003		end of text; end of message
004		end of transmission
005		enquiry; who are you?
006		acknowledge; are you ...?
007		ring bell
008		backspace; format effector
009		horizontal tab
010		line feed; line space
011		vertical tab
012		form feed to top of next page
013		carriage return to beginning of line
014		shift out
015		shift in
016		data link escape
017		device control 1
018		device control 2
019		device control 3
020		device control 4
021		negative acknowledge; error
022		synchronous idle
023		end of transmission block; logical end of medium
024		cancel
025		end of medium
026		substitute
027		escape
028		file separator
029		group separator
030		record separator
031		unit separator
032		space
033	!	
034	"	
035	#	
036	$	
037	%	
038	&	
039	'	apostrophe; replaced by acute accent (´) on some installations
040	(	
041	)	
042	*	
043	+	
044	,	
045	-	hyphen or minus sign
046	.	
047	/	
048	0	numeral zero
049	1	
050	2	
051	3	
052	4	
053	5	
054	6	
055	7	

Character Code	Graphic (if any)	Remarks
056	8	
057	9	
058	:	
059	;	
060	<	
061	=	
062	>	
063	?	
064	@	
065	A	
066	B	
067	C	
068	D	
069	E	
070	F	
071	G	
072	H	
073	I	
074	J	
075	K	
076	L	
077	M	
078	N	
079	O	
080	P	
081	Q	
082	R	
083	S	
084	T	
085	U	
086	V	
087	W	
088	X	
089	Y	
090	Z	
091	[	
092	\	
093	]	
094	^	circumflex; also vertical arrow (↑) on some installations
095	_	underscore; also horizontal arrow (←) on some installations
096	`	grave accent
097	a	
098	b	
099	c	
100	d	
101	e	
102	f	
103	g	
104	h	
105	i	
106	j	
107	k	
108	l	
109	m	
110	n	
111	o	
112	p	
113	q	
114	r	
115	s	
116	t	
117	u	
118	v	
119	w	
120	x	
121	y	
122	z	
123	{	
124	|	
125	}	
126	~	
127		delete; rub out

Notes:

1) For control characters (codes 0 through 31, and 127) the definitions under "remarks" are those given by ASCII. The actual effect of a given control character depends on the device to which it is sent.

2) On many installations, only the upper case alphabet can be input or output. If the lower case alphabet is available, lower case letters may be used in CS-4 identifiers, where they are considered distinct from upper case ones. For example, any of these three names could be declared as a variable:

```
DISTANCE
Distance
distance
```

If all three were referred to, they would be interpreted as different names by the compiler. Lower case letters may not be used in place of upper case ones in words which have a special meaning in CS-4 (including reserved words, words with special-context meanings, and names of pre-defined functions and operators).

APPENDIX B

PRECEDENCE AND ASSOCIATIVITY OF PRE-DEFINED OPERATORS USED IN THIS VOLUME

Operators	Precedence*	Associativity
**	10	right
+ (prefix), - (prefix), ~, NOT	9	--
*, /, IDIV	8	left
+ (infix), - (infix)	6	left
||	5	left
=, ~=, >, <, >=, <=	4	left
&, AND, NAND	3	left
|, OR, XOR, NOR	2	left
:=	1	right

* Numbers in this column indicate relative precedence -- operators with higher numbers have higher precedence than those with lower ones. (The numbers themselves are taken from the full table on page 71 of the Language Reference Manual.)

Reader's Comment Form

A CS-4 PRIMER

Your comments on this Primer are encouraged, so that we may make improvements to subsequent editions. Opinions on the Primer's usefulness and readability, suggestions for additions and deletions, and notes of specific errors and omissions are all welcome.

COMMENTS

cut along line

Name: _____

Organization or Company: _____

_____

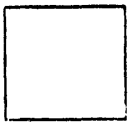
Address: _____

_____

_____

Date of latest update incorporated in your Primer: _____

fold



Mr. Warren E. Loper
Code 5200
Naval Electronics Laboratory Center
San Diego, California 92152

fold

Reader's Comment Form

A CS-4 PRIMER

Your comments on this Primer are encouraged, so that we may make improvements to subsequent editions. Opinions on the Primer's usefulness and readability, suggestions for additions and deletions, and notes of specific errors and omissions are all welcome.

COMMENTS

cut along line

Name: _____

Organization or Company: _____

_____

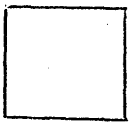
Address: _____

_____

_____

Date of latest update incorporated in your Primer: _____

fold



Mr. Warren E. Loper
Code 5200
Naval Electronics Laboratory Center
San Diego, California 92152

fold

cut along line

Update Request Form

A CS-4 PRIMER

It is planned that updates will be issued to the Primer on an "as required" basis. If you wish to be placed on the distribution list for these updates, please complete this form and mail it to the address on the reverse side.

Name: _____

Organization or Company: _____

_____

Address: _____

_____

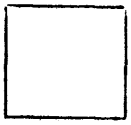
_____

Telephone: _____

Date: _____

Date of Latest Update Incorporated in Your Primer: _____

fold



Mr. Warren E. Loper
Code 5200
Naval Electronics Laboratory Center
San Diego, California 92152

fold

Change of Address Form

A CS-4 PRIMER

If you are currently on the distribution list for updates to this Primer and your address changes, please complete this form and mail it to the address on the reverse side:

Name: _____

Address presently on the distribution list:

_____

_____

_____

New Organization or Company: _____

_____

New Address: _____

_____

New Telephone: _____

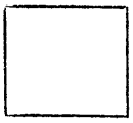
Effective Date: _____

Today's Date: _____

Date of Latest Update Incorporated in Your Primer: _____

cut along line

fold



Mr. Warren E. Loper
Code 5200
Naval Electronics Laboratory Center
San Diego, California 92152

fold