A FOCAL PRIMER

2nd Edition

With Chapters on BASIC
and FORTAN IV with WATFOR

Howard C. Howland

A FOCAL PRIMER

A FOCAL PRIMER

2nd Edition

With Chapters on BASIC
and FORTAN IV with WATFOR

Howard C. Howland

Algebraic Languages
Ithaca, New York

Algebraic Languages, Box 113
Ithaca, New York

Current printing (last digit):
9 8 7 6 5 4 3 2

Printed in the United States of America

To the fond memory of my colleague, Frank Rosenblatt, who learned FOCAL one day to check an astronomical calculation concerning the detection of planets about other suns.

A FOCAL Primer

CONTENTS

Preface

This manual was written for the students of my own course on interactive computing, Biological Sciences 106, and for users of the Division of Biological Sciences Interactive Computing Facility at Cornell University.

In this second edition, I have incorporated material on other algebraic languages, particularly BASIC and FORTRAN, and have expanded the exercises. The text has been retyped and hopefully most of the errors of the first edition have been removed.

I am indebted to Mrs. Sherry Murray for preparing a camera-ready typescript, as well as for much editorial help, and to my wife, Monica Howland, for executing the figures and cover drawings.

ERRATA

Page	Correction
9	$\sqrt{X}$ FSQT(X)
12	...by writing TYPE "A", A or TYPE ?A?
20	...and (3,5); (2.3,0.3) and ...
23	Ø1.1Ø ASK N,!; IF ...
39	VA(R+RM*C)
64	16Ø PRINT "Ø","Ø", "1","Ø","INF"
65	2Ø FOR I = 1 TO N
81	FOCAL 69 gives +1 as the value of FSGN(Ø)
97	Arctangent

CHAPTER I

INTRODUCTION: INTERACTIVE COMPUTING IN EDUCATION TODAY

Of all the technological developments in this most technological of all centuries, electronic digital computing will surely have a profound influence on the future course of human affairs. Forming the very heart of automation, the programmed digital computer threatens to replace a large fraction of the brain power of our industries.

Parallel to this social change, are to be anticipated profound alterations in the field of mathematics, where the economics of problem-solving has already been turned upside down by the power and speed of digital computers.

In particular, the miracle of electronic computing methods has permitted the economic solution of specific cases of equation systems which were formerly effectively insoluble, and has concurrently reduced the value of general, closed-form solutions. To the university student, this means that he or she is likely to see less of the classical calculus and more of the numerical analysis in future mathematics curricula.

But what of today's students? University curricula have enormous inertia, and present day digital computers are often shielded from their users by an array of counters, key-punch operators, account numbers, and complex software systems.

The picture is particularly acute in view of the changed pattern of federal funding of research. Educational computing has always been a stepchild of research computing since its very inception, and research computing in our nation is decidedly in trouble. In this dark financial climate, how can today's students economically learn digital computing?

I believe that interactive computing on a mini computer provides an economic answer in the coming decade.* Accordingly, this book is about a "mini" computer language, FOCAL, which was written by Richard M. Merrill of the Digital Equipment Corporation for the PDP-8 series of computers. FOCAL is a powerful and elegant interactive language of the FORTRAN branch of the Indo-European computing language tree (BASIC is another language on the same branch; on the other hand, APL, yet another interactive computing language, quite unlike FORTRAN, FOCAL, or BASIC, is not).

FOCAL can be learned in a course, with a tutor, or all by oneself while sitting at a computer teletype. Programs in FOCAL can allow one to add up a checkbook, calculate an average, solve a set of differential equations, model a population, compute the University's budget, or play tic-tac-toe. Virtually any mathematical task may be programmed in FOCAL, provided it is broken down into bite size chunks that the computer can digest.

---

*A mini computer in the current jargon has 4 to 8 thousand 12 to 16 bit words of core storage.

Because FOCAL is interactive, a program written in it can be tested and "debugged" very rapidly. Users of batch process languages (who submit a deck of computing cards to a computing center and get them back later together with printed program output) are accustomed to learning, hours after they have submitted a job, that the program has failed to run because they have forgotten a comma. Users of an interactive language get diagnostic correction comments in seconds.

Because mini computers are indeed small, they cannot remember much at one time (FOCAL in a four-user configuration on an 8,000 word machine allows about 800 12 bit words per user as opposed to 10,000 words for a normal batched FORTRAN job run under the WATFOR compiler). A FOCAL user can write a program of maximum size equivalent to about fifty FORTRAN statements with twenty to thirty stored variables. An attempt to store a larger program or more variables would overload the computer's memory.

This size restriction may appear to be a severe one to the experienced FORTRAN programmer who is accustomed to handling great arrays in large amounts of core storage. But to the average student, even the average researcher, for most programs this restriction may not be serious. This is because many large jobs can indeed be broken into small ones and the very rapid "turn around" which interactive computing provides, often far outweighs the space restrictions of the "mini" computer.

It will be interesting to see what languages and what machines dominate the academic computing scene in the next ten to fifteen years. It is my belief that a significant portion of academic computing will be done on small time-sharing computers, whose high speed, low cost and ready accessibility will give them an economic advantage over a large centralized system. If this is true, then languages such as FOCAL and BASIC will play a key role in computer education.

In the following chapters I have attempted to emphasize the general aspects of algebraic computing languages. The beginning student may be disheartened by the multitude of languages he encounters and wish that there was one simple algebraic language period. He is advised to take heart! The differences are not as severe as they seem. Indeed I believe that the major outlines of all future algebraic languages can be discerned in the languages that are extant today, and it is these common fundamental elements that I have attempted to emphasize in this treatment of FOCAL. In addition I believe that the student will find that FOCAL affords an optimal way into the diversity of computer languages that are in current use.

Some Words About FOCAL in Particular

FOCAL has several advantages over other interactive computing languages, and, even though the student is not yet familiar with FOCAL, this is perhaps a good place to cite these advantages by way of encouragement.

FOCAL is easy to debug. To "debug" a program is to find errors in it. Because FOCAL has direct commands which let a user access any part of his stored program, it is particularly easy to locate a trouble spot in a program. One command in FOCAL allows one to have the computer type out all the values of all his stored variables by simply typing TYPE $ and then a carriage return.

FOCAL has powerful editing features which allow the user to correct an error easily once it is found.

FOCAL is a fast computing language. A program written in FOCAL may run in two thirds the time that an equivalent BASIC program will run on the same or comparable machine.

FOCAL has one of the most elegant subroutine capabilities of all algebraic languages in the "DO" statement.

FOCAL is a compact language due to the fact that its commands may be abbreviated and that several commands may be written on one line.

All of these features make FOCAL a worthy competitor of BASIC; and in many situations the additional economy of FOCAL may tip the scales well in its favor.

CHAPTER II

DIRECT COMPUTATION AND ELEMENTARY STORED PROGRAMS

Introduction

There are many algebraic computing languages currently in use around the country. Some of them are given in Table 1.

Table 1. Some Algebraic Computing Languages

Language	Interactive* or not	Availability
FORTRAN	no	Universal on large machines
ALGOL	no	On many large machines
APL	yes	On IBM machines
BASIC	yes	Most available of all interactive languages
FOCAL	yes	On PDP-8 computers
CUPL	no	At Cornell on 360/65

*Programs of interactive languages can be modified by the programmer from a console or a teletype while the program is in the computer, thus greatly facilitating correction of errors in the program.

The advantages of FOCAL as an interactive language have already been cited above. The question of an interactive language versus a batch-processed language, however, is one which deserves mention here.

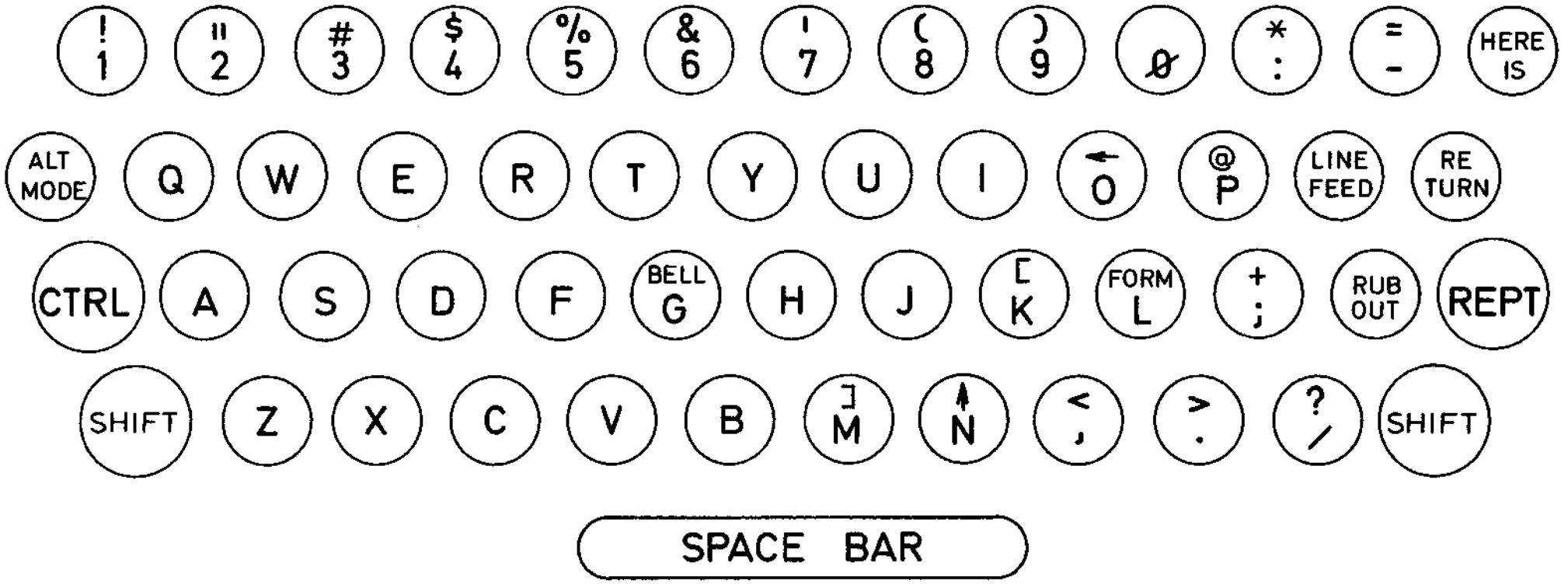

! 1
" 2
3
$ 4
% 5
& 6
' 7
(8
) 9
0
* :
= -
HERE IS
ALT MODE
Q
W
E
R
T
Y
U
I
O
@ P
LINE FEED
RE TURN
CTRL
A
S
D
F
BELL G
H
J
K
FORM L
+ ;
RUB OUT
REPT
SHIFT
Z
X
C
V
B
M
N
< ,
> .
? /
SHIFT
SPACE BAR

Figure 1.

Batch processed languages are adapted to the card input, line printer output configuration which large computers started with in the late 1950's. That configuration is admirably saving of machine time, and, in most circumstances, terribly wasteful of human time. I say "in most circumstances" because occasionally the "turnaround time", the time between submission of a program to a computing center and the receipt of the executed program's output, is short enough, on the order of a few minutes, to make the batch process worthwhile for learning programming. But in ordinary circumstances the user of a batch processed language will wait for several hours only to learn that he has forgotten a comma somewhere in the program and for that reason it will not run.

Students can be trained to put up with anything, particularly if no alternative is offered them. However, with the advent of interactive languages, there is little question that, unless a central computing facility offers remarkable turnaround time, (as some universities do with "cafeteria batched", WATFOR compiled, FORTRAN), the only reasonable choice for learning an algebraic computing language is to use an interactive computer.

Moreover, my experience has been that once a student has mastered any of the algebraic computing languages he will be able to learn any other easily by self study.

Direct Computation in FOCAL. The TYPE Statement and Algebraic Expressions

Assume you are sitting at the teletype of a PDP-8 computer whose keyboard looks like that of Figure 1. The computer has been loaded with FOCAL and is ready to go. The computer indicates that it is in a state ready to accept commands by typing an "*". (The details of actually turning on the computer, loading FOCAL, or any other odds and ends are best learned while you are actually in front of the computer.)

In order to compute an algebraic expression such as

$$(3.14159)\ (\ \sqrt{(42/31 + 26.9)}$$

one types the following on the teletype:

```
TYPE (3.14159) * FSQT( (42/31) + 26.9)
```

and then depresses the carriage-return key at the right of the teletype keyboard. The computer then types the answer:

```
16.69*
```

or perhaps

```
Ø.166992E+Ø2*
```

The answer in this latter case is the number preceeding the E times 10 raised to the power following the E.

Thus: .105621E05 = 10562.1

.635292E-03 = .000635292

This "exponential", or "scientific" notation is almost universal in computing, and the student will do well to memorize its meaning. One obvious advantage is that very large and very small numbers can be expressed in about 12 characters of type to six places of accuracy.* It might be noted here that FOCAL works only to six places of accuracy and can represent numbers in exponential notation as large as .999999E619.

A method for designating the format of the answer (whether it is in exponential or decimal notation and how many decimal places, etc., are desired) is given below.

Algebraic Expressions

Most algebraic expressions have their counterpart in FOCAL. Table 2 gives some common ones. A complete list of functions is supplied in Appendix 1. Note that no space is allowed between the function name and the first parenthesis of the argument.

Errors

You may be wondering what happens if you ask FOCAL to execute an impossible command. Well, "ask a silly question..." and you get out an error code, like "?07.38". Appendix 3 gives the meaning of these error codes which are designed to help you correct your mistake. Usually the error code will clear the matter up immediately, but occasionally it won't. If you cannot understand what is wrong with a statement, try retyping the offending statement paying particular attention to spaces and punctuation before you call for help. Further "debugging" aids are discussed in Chapter 5.

Ambiguous Expressions and Rules of Precedence

An expression like 3·4+2/3 is ambiguous unless you know the order in which the operations in it are performed in its evaluation. It might be

$$(3\cdot 4)+(2/3) = 12.66$$

or $$(3\cdot(4+2))/3 = 6$$

or $$3\cdot(4+(2/3)) = 14$$

In every case these ambiguities can be surmounted by using parentheses and I recommend their use. As a matter of fact, however, FOCAL evaluates arithmetic expressions in a fixed way. It performs operations in a precedence order: first exponentiation, then multiplication, then division, then subtraction and addition. Expressions of the same precedence are evaluated from left to right. According to these rules the expression 3·4+2/3 would be equal to 12.666.

FOCAL recognizes parentheses, square brackets and angular brackets all as parentheses and uses them to evaluate expressions. You will note that parentheses always occur in pairs.

---

*on PDP-8 machines.

Table 2

Mathematics	FOCAL	Note
X	FSQT(X)	Argument must be ≥ 0.
$\sin X$	FSIN(X)	Argument is in radians ($360° = 2\pi$ radians).
e^X	FEXP(X)	
$\text{Log}_e x$	FLOG(X)	
$\text{Log}_{10} x$	FLOG(X)/FLOG(10)	This follows from the definition of logarithms.
A^X	FEXP(FLOG(A)*X)	For 10^X set A = 10
X^4	X↑4	Exponents restricted to positive integers. (No warning if exponent is fractional; it is simply truncated.)
X/Y	X/Y	
XY	X*Y	A common error is to forget the "*" in multiplication.
$X+Y$	X+Y	
$X-Y$	X-Y	

Exercise Set 1.

Pencil and Paper

1. Write the following numbers in exponential notation:

27.3	-240
100	1/2
1.2	pi

2. Write the following numbers in ordinary decimal notation:

.141416E01	-.829000E03
.123456E-02	.200000E10

3. Write FOCAL statements for the following algebraic expressions or their equivalent:

   a) sin((2/360)(45))
   b) [(3.6)(48.2)] .30103
   c) 26(45+32+10)/627
   d) cos(45°)
   e) 27.302
   f) $\log_{10} 34.2$
   g) $(66^2+27^3)/284$

Computer

4. Evaluate the expressions in 3 above on the computer. Hand in your complete computer output.

5. Perform a set of computations to show that the rules of precedence given above are, in fact, those which the computer uses. Append a discussion telling how your computations do this.

Indirect Computation: Elementary Programs in FOCAL

Instead of directly typing out the values of expressions, we can store these values for further reference by setting a variable equal to the value of the expression using the FOCAL command SET .

For example, we may say:

```
SET X = 43/2+FSQT(4.92)
```

When this command is executed, X is assigned the value of the right hand side of the equation. The general name for a SET command is an "assignment" command.

In addition, we may use variables as well as numbers on the right hand side of a SET command, provided values have previously been assigned to these variables. A value may be assigned to a variable by means of a SET command or an ASK command in which the value must be typed in from the teletype keyboard.

These two features constitute the first level of flexibility in handling numbers in the computer. This section of the primer will describe the rules in FOCAL for getting numbers into and out of the computer and elementary programs involving these numbers.

The following simple example will illustrate the elements of a program and give a background for the rules which are presented later:

Consider the computation of A from X and Y in the two equations:

$$Z = (X^2+Y^2)^{1/2} \qquad (1)$$

$$A = (Z/100) + 27 \qquad (2)$$

Suppose further that we wish to perform this computation for a number of pairs of X and Y. We may do this by first writing a stored program and then executing it. The program might be:

```
Ø1.Ø5 C PROGRAM EXAMPLE 1.1
Ø1.1Ø A X,Y
Ø1.2Ø SET Z = FSQT((X↑2)+(Y↑2))
Ø1.3Ø SET A = (Z/1ØØ)+27; TYPE A
```

Each statement would be entered into the program by typing it on the teletype and depressing the RETURN key. After each statement the computer would return the carriage, advance a line and type an "*" to indicate that it awaited your next command. After the last statement of the program was entered it could be executed by typing GO and depressing the RETURN key. The statements would be executed in numerical order. Since statement 1.05 is a "dummy" comment statement (because it begins with a "C"), the computer will pass over it and execute statement 1.1. It will type a colon and wait for you to type in the value of X. This is done by simply typing a number and a space, comma, or carriage return. The computer will then ask for the value of Y in the same way. It will then execute statements 1.2 and 1.3, in the first case using the values of X and Y supplied as input, and in

the second using the value of Z it has just computed. The last part of the compound statement 1.3 causes the computer to type out the value of A it has just computed.

To perform the same calculation for another pair of values of X and Y you would type GO , depress the return key and the same sequence would be followed again.

Comments on Program 1.1

Statement number	Comment
1.05	This is a non-executable statement. It simply serves to label the program. The computer skips over any statement beginning with a C.
1.1	This statement asks for values of the variables X and Y. If desired, one may have the name of the variable asked for printed before the colon by writing ASK ?X?, ?Y?.
1.2	This statement computes the value of an intermediate variable Z. If desired this value could also be printed out in a type statement, e.g., 1.4 TYPE Z .
1.3	This statement computes the value of A. The second part of this compound statement after the semicolon causes A to be printed out. If desired, the name of the variable could also be printed out either by writing TYPE "A", "A" or TYPE ?A? . The reader is invited to experiment with both these methods on the computer.

It should be noted that after the program has been executed the value of Z is still in the computer. This value may be examined after the program has been executed by typing on the teletype:

```
TYPE Z ,
```

and depressing the carriage return.

Elements of a Program

We shall now preceed to give the rules for using the statements presented above.

Statement numbers

Each program statement in FOCAL must be preceded by a statement number. State-

ment numbers are decimals between 1.01 and 31.99 (in FOCAL-8, 1969). The values 1.0, 2.0,...31.0 are reserved for a special use in referring to all statements of a section (see below). FOCAL executes indirect statements in statement number order unless a logical command alters that sequence.

The user need not write two decimal places, e.g., 1.1 will serve for statement 1.10. More importantly, the user need not enter statements in statement number order. Regardless of how they are entered FOCAL will execute them in correct order.

Assigning Variable Names

All variable names must begin with a letter and are either one letter or one letter and another letter or number (i.e., they are one or two characters long). Longer names may be used but FOCAL will recognize only the first two characters. The letter F is excluded as a beginning for a variable name. No distinction is made in FOCAL between integer and "floating point" variables; all numbers are recorded internally in exponential notation, though (as may be seen below) they may be printed as integers.

The SET Statement

The statement SET X=Y where X is a variable name and Y is an algebraic expression causes the value of Y to be computed and assigned to the variable X. One peculiarity of this operation is that the variable name X may also appear in the expression, Y, in a way which may at first appear contradictory, e.g., the expression:

```
SET X = X + 1
```

is legitimate and causes the current value of X to be augmented by 1 whenever the statement is executed. In general, variables on the right hand side of a SET statement always have the values they had before the set statement was executed.

Comments

Comments may be included in your program for your own reference. A comment statement receives a statement number just as an ordinary instruction; however, it is ignored by the computer in execution. A comment statement is simply the letter C followed by the text you wish to write. A comment statement is only printed out when the program or statement is written out with a WRITE command (see below). In order to have text printed along with numbers in response to a TYPE command, another technique is used (see below).

Data Input

Data may be entered into an indirect program in two ways. Either the data is included in an indirect statement like:

```
1.1 SET PI = 3.14159
```

or it is asked for by the program itself in an ASK statement, e.g.,

```
1.1 ASK PI
```

When the computer, in executing its program, comes to an ASK statement, it types a colon and waits for that value to be entered from the teletype, either by typing or by reading from the teletype reader. Whenever the computer types a colon it indicates that it is waiting for a response. It will accept the following:

a) a number perhaps preceded by spaces and certainly followed by a space, comma, or carriage return;
b) a carriage return, in which case it assumes the number entered was zero;
c) depression of the "ALT MOD" key, which it takes as an instruction to retain the previous value of that number.

According to the response, the computer assigns a value to the variable mentioned in the ASK statement. It should be noted that several variables may be entered by one statement; e.g., ASK, A,B,C,D will cause the computer to ask for values, each time typing a colon, waiting for a response and assigning that value to the proper variable reading left to right.

Ways to Tell the Name of the Variable Being Asked

Text may be printed out by an ASK statement if it is enclosed in quotation marks. Thus one might write:

```
1.15 ASK "ENTER RADIUS", R
```

and when this statement was executed the computer would type

```
ENTER RADIUS:
```

and wait for the user to supply a value of R.

Another way to have the computer specify the variable name that it is asking for is to enclose the name in question marks. Thus

```
1.15 ASK ?R?
```

will cause the computer to print

```
R:
```

and wait for the user to supply a value of R.

The use of question marks in this fashion is called turning on and off the trace feature of FOCAL. When in the course of execution the computer encounters a question mark it will begin to print out all the commands as it executes them. When it finds a second question mark it will stop doing this. This feature is often useful in debugging a program (see below) but here it is simply used to cause the printing out of variable names as they are asked for. If you use this method of specifying input variables be sure that you use the question marks in pairs! Otherwise you may find that the computer is printing out your entire program.

Data Output

The value of a variable is typed out by use of the TYPE command. In the simplest case the number is printed out in response to a command of the following form:

```
1.1 TYPE A
```

However, we may wish to control the format of a number or add text to accompany the number to describe its meaning. A mark of a good programmer is often thought to be the readability of his computer output. In FOCAL output formating is very straightforward compared to many other computer languages and is described in its entirety below.

The Format of Numbers in Output

We have already discussed the exponential notation for numbers above. It is often of use to be able to specify exactly how a number will be printed out on the computer, for while the exponential format is one in which all the accuracy of the computer will always be expressed, nonetheless it is tedious to read, and this may lead to inaccuracies itself. Hence there are ways of changing the output format of numbers in FOCAL, and the computer always prints out numbers according to the most recent format instruction it was given. The format instruction for the statement TYPE A may be given by interposing a % ____, sign where the blank may or may not contain a number indicating where to place the decimal point and how many places are to be printed. The simplest way to explain the command is with examples, and these are given in Table 2.

Table 2. Formats for Numerical Output

Number, X	Desired appearance	Command with Format
61.703	.617030E+02	T %,X
	61.7	T %3.01, X
	61.70	T %4.02, X
	62	T %2.00, X or T %2, X
	62	T % 6, X

Note that FOCAL inserts leading blanks before the number if the format specified has more places than the number to be printed. The reader may verify that it also inserts following zeros after a decimal point.

Other Formating: Text and Carriage Control

Much time is wasted in formating output in non-interactive computing languages because the commands are so arbitrary. In FOCAL the user can easily format his output by cut and try methods.

The other three formating symbols are quotation marks which, when used in as ASK or TYPE statement, will cause the typing out of all that is enclosed between them; the exclamation mark, !, which causes a carriage return and line feed, and the number sign, #, which causes a carriage return with no line feed.

Some of these symbols might be added to the program of 1.1 as follows:

```
1.Ø5 T "PROGRAM 1.2",!
1.1 ASK "ENTER VALUES OF X AND Y",X,Y
⋮
1.4 TYPE ! , "THE ANSWER IS" , A,!
```

The reader is invited to experiment with formating the output of his programs in a clear and legible fashion.

Correcting Indirect Statements and Erasing Programs

Any indirect statement may be erased by typing ERASE followed by the statement number, e.g., ERASE 12.34, and depressing the carriage return. A whole section may be erased by typing ERASE and the section number, e.g., ERASE 3.0, and depressing the carriage return.

The entire program may be erased by typing ERASE ALL and depressing the carriage return. The command ERASE alone, will cause all variables in the internal symbol table to be set to zero and the variable names erased. To inspect this command's effects, type TYPE $ and return carriage.

Any statement may be changed by simply retyping and re-entering it (e.g., typing the statement and depressing carriage return).

If an error in typing is made, the last character typed may be erased by depressing the RUBOUT key immediately after typing the incorrect character. N preceding characters may be erased by depressing the rubout key N times. The entire statement line may be erased by typing ← before the carriage return key is depressed.

Setting Variables to Zero with the ERASE Command

The ERASE command used in a direct or indirect statement with no numbers following it may be used to set the values of all variables to zero. It is often used at the start of a program to insure that all of the variables which are used as accumulators are zero. A typical use might be to begin a program with the statement:

```
1.Ø5 ERASE
```

The MODIFY Command

A line may be altered with the MODIFY command. Suppose it is desired to change all the X's to Y's in the following statement:

```
6.3 SET Z = X↑2 + 2*X*A + X/3
```

One types:

```
MODIFY 6.3
```

and depresses the carriage return. The computer then silently waits for the user to type a "search character". In our example the user would type X and the computer would respond by typing

```
SET Z = X
```

and stop. The user would depress the rubout key, type Y, and then depress the CONTROL and FORM keys to cause the computer to continue typing the statement until the next X was encountered. The user would delete this in the same way and proceed as before until all the X's were found and deleted. At the end of the line the computer will return the carriage. The modified statement has replaced the original in the computer's memory, as could be verified by typing WRITE 6.3.

In order to change search characters in mid-stream the user may type CONTROL BELL, enter the new character and the computer will search the rest of the line for it.

If at any time before the entire line has been searched the user depresses the carriage return key, the rest of the line will be erased.

Until the user becomes confident in his usage of the MODIFY statement, it is recommended that he check all of his modified commands by having the computer write them out after he has modified them. The user must also beware of inadvertently entering an indirect command in the confusion of thinking he is still modifying another command, or, like the broom in the hands of the sorcerer's apprentice, the computer will appear to have a mind of its own.

Direct Use of the DO Command to Check Individual Indirect Statements

All FOCAL commands may be executed directly as well as being executed indirectly. Because this is so, it is possible to jump into the center of an indirect program and test its individual statements. This is accomplished with the help of the DO command as follows: Suppose we wished to test statement 1.2 in program 1.1 above which had already been entered into the computer, we might type the direct command:

```
SET X = 4; SET Y = 3; DO 1.2; TYPE %, Z
```

and execute it by depressing the carriage return. The computer will type

```
.5ØØØØØE+Ø1
```

if the statement is correct.

Entering Statements, the Asterisk and Carriage Return

We have already noted above that when the computer is ready to receive commands it types an asterisk. The user may then type any commands he desired, but these commands are not entered until he depresses the carriage return (hereafter signified by "c.r."). An indication that FOCAL is working correctly is it responds with an asterisk to any carriage return other than that following a data entry.

Compound Commands

It will have been noted above that commands may be compounded with a semicolon separating them as in:

```
ASK A,B; TYPE A/B↑2,!
```

As many separate commands as will fit in one line may be entered in this way, and they will be executed from left to right. This feature, unusual in an algebraic computing language, is particularly helpful in FOCAL in saving space in the computer's memory.

Abbreviations

All commands may be either typed out such as ASK, SET or abbreviated with a single letter. A complete list of abbreviations is given in Appendix 2. As an example here, the statement ASK A,B; TYPE A/B↑2,! may be written A A,B; T A/B↑2,! With a little practice it becomes as easy to read the abbreviations as the extensive form of the commands. Again, this is a feature designed to help save memory space.

Writing Out of Stored Programs

FOCAL will write out an entire program in statement number sequence if the command WRITE is typed out on the teletype and the carriage return is depressed. FOCAL will write an individual section (e.g., all statements beginning with 3) if the command WRITE 3.Ø or WRITE 11.Ø is given.

Punching Out Stored FOCAL Programs on Paper Tape

FOCAL programs may be punched out on paper tape for future use. This is done by the following steps:

1) Type WRITE on the teletype; do not depress the carriage return key.
2) Turn on the paper tape punch by depressing the ON button, depress the keys SHIFT, REPEAT, and P in that order. This will cause the printing of a string of @ symbols which function as leader tape for the program. After about a half a line of @'s have been printed:
3) Remove fingers from the P, REPEAT and SHIFT keys in that order.
4) Depress the carriage return key. This will cause the program to be printed and punched simultaneously.
5) After the computer has stopped printing, generate another string of @'s

for follower tape by the same method in steps 2 and 3 above.

6) Turn off the paper tape punch.

Entering Stored Programs from Paper Tape

The computer cannot distinguish the paper tape reader from a very fast and regular typist. Hence programs may be entered into the computer from paper tape by simply inserting the paper tape into the reader and turning it on. (Note that when the reader switch is set to FREE the paper tape may be positioned by hand by simply pulling on one end or the other.)

Caution!

In order to make sure that you do not load your program on top of another, be sure to type ERASE ALL to get rid of anything left in the computer before you load your program.

Some computer configurations require that the "keyboard echo" be turned off when the tape is read in. This will cause the tape to be read in without the program being written on the computer (and a resultant jamming of the input buffers of the computer). To turn off the keyboard echo one usually depresses the CONTROL and R keys simultaneously. To restore the keyboard echo after the tape has been read in one depresses the CONTROL and C keys simultaneously.

After the tape has been read in the user should set the switch to the FREE position.

Exercise Set 2.

Pencil and Paper

1. Write a program which asks for the lengths of the sides of a right triangle, A and B. Have it compute and print out the area, AR.

2. Write a program to ask for the weight of five objects and to compute and print out their average weight.

3. The weight of a particular species of animal is given by the formula, $W=.367\ L^3$, where W is the weight in grams and L is the length of the animal in centimeters. Write a program which asks for the length of the animal and computes and prints out its weight.

4. The kinetic energy of a moving object is $1/2\ MV^2$ where M is the mass of the object and V is its velocity. Write a program to compute the kinetic energy of an object, given its mass and velocity.

5. The following sequence of commands is given the computer:
   S Y = 2; S X = 3; S Z = X↑2+Y; T % 5.03, Z; S X = Z;T %,X,! c.r.
   What will the computer do in response to them?

Computer

6. Load the following program into the computer:

```
Ø1.Ø5 C EXERCISE SET 2, QUESTION 1
Ø1.1Ø A X1, Y1,X2,Y2,!
Ø1.2Ø S L = FSQT((X1-X2)↑2  +  (Y1-Y2)↑2)
Ø1.3Ø T  "DISTANCE",%,L,!
```

   Use it to compute the distance between the following pairs of points: (5,7) and (3,5); (2,3,.3) and (.8,4); (10,1) and (0,0).

7. Modify the above program to compute and print the distance L/100.

8. Type in and execute the programs of the pencil and paper exercise above.

9. Save one of your programs on paper tape. Erase the computer memory, verify that it is erased and reload your program from paper tape. Try it out to see if you have reloaded it successfully.

CHAPTER III

FLOW CHARTING AND STATEMENTS WHICH ALTER THE SEQUENCE OF COMPUTATION

Introduction

Many calculations require repetitive performance of one sequence of operations again and again; others require alternate procedures depending upon what the value of a particular variable may be. Still others require that operations be performed on all members of a particular set of numbers. All of these eventualities may be handled in FOCAL with the use of DO, IF, FOR and GOTO statements. All algebraic languages have similar statements and one universal notation for such logical operations is given by a flow chart of the single operations. Such flow charts and the logical statements they represent are discussed in this chapter.

The IF Statement, Avoiding Division by Zero

Often we must divide numbers in a program with the possibility that we will encounter a zero denominator, in which case the answer would be undefined, and an error code would be typed, after which the computer would stop. Of course, the computer could be programmed to plug in any old non-zero value for a zero denominator and go right on, but division by zero is viewed as such an important error that the user must be informed immediately of it.

To avoid this we may use an IF statement to halt division by zero before it occurs. For example:

```
Ø1.Ø5 C PROGRAM EXAMPLE 2.1
Ø1.1Ø ASK X,Y,!
Ø1.2Ø IF (Y) 1.3,1.4,1.3
Ø1.3Ø T %6.Ø2, X/Y,!;GOTO 1.1
Ø1.4Ø T "DIVISION BY ZERO REQUESTED"
```

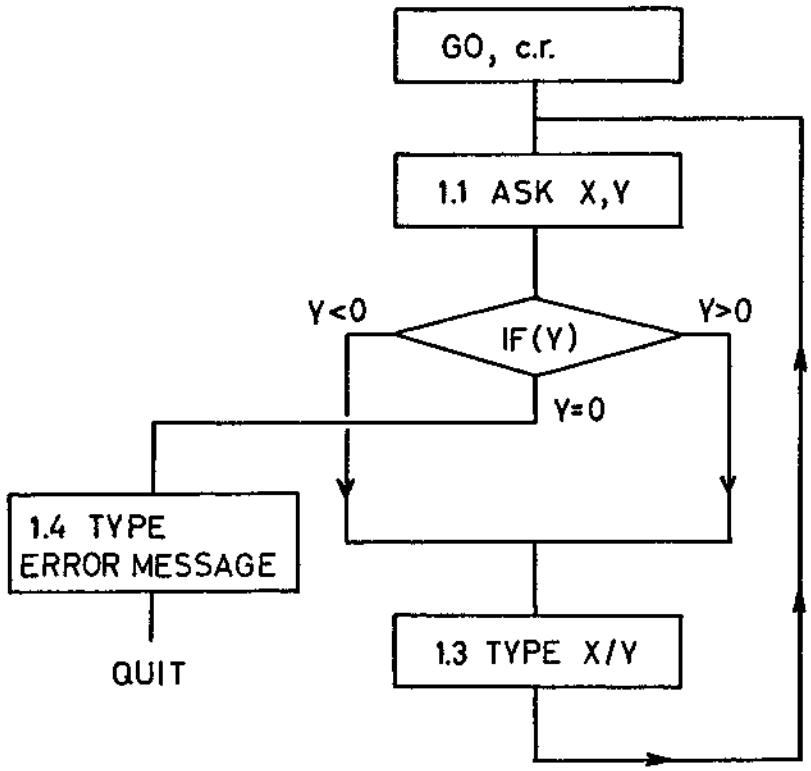
This program contains two new statements: the IF statement in 1.2 and the GOTO at the end of 1.3. Their effect may be seen in Flow Diagram 1.

One reads such a diagram by simply following the arrows. Starting at the top with GO c.r., the program asks for the values of X and Y. The IF statement then tests the value of Y. If it is either greater or less than (but not equal to) zero, control is passed to statement 1.3 and the value of X/Y is typed out. The GOTO 1.1 statement then returns the program to statement 1.1 and new values of X and Y are requested.

If ever a zero value of Y is encountered after an ASK statement, control passes to statement 1.4 which causes a warning to be typed out. After 1.4 the program terminates for lack of a higher numbered statement.

Flow Diagram Conventions

Conventions in flow diagrams vary somewhat but are usually self-explanatory. The IF statement is usually written in a diamond, ordinary statements in rectangles.



Flow Diagram 1.

You will meet a few more notational conventions below.

The IF Statement in FOCAL

The IF statement takes as its argument any arithmetic variable or expression. Its usual form is:

IF (argument) a,b,c

where the argument is a variable or expression and a,b and c are statement numbers of statements which will be executed next if the argument is less than zero (a next) equal to zero (b next) or greater than zero (c next). Upon transfer of control to either a, b, or c the sequence proceeds as usual to the next highest statement number after a, b, or c, respectively. In general (with but one or two exceptions in FOCAL), when control is transferred to a new statement number, no note is taken of how the program got to that statement--it simply proceeds to execute in normal statement number order.

Shortened forms of the IF statement are:

IF (arg.) a,b; statement I

and

IF (arg.) a; statement II

In I, the statement will be executed only if arg. is greater than zero; otherwise control will pass to a, or b, depending upon if arg. is less than or equal to zero. In II, the statement will be executed if arg. is equal to or greater than zero, and otherwise the control will pass to a.

The following program illustrates further the use of the IF statement. It sums the positive numbers of a set of N numbers but excludes negative numbers from the sum.

```
Ø1.Ø5  C  PROGRAM EXAMPLE 2.2
Ø1.1Ø ASK N!; SET J = 0;  SET SU = 0
Ø1.2Ø ASK X,!;  IF (X) 1.4, 1.3, 1.3
Ø1.3Ø SET  SU = SU + X
Ø1.4Ø SET J = J+1
Ø1.5Ø IF (N-J) 1.6, 1.6, 1.2
Ø1.6Ø TYPE %, SU,!
```

The program is diagrammed in Flow diagram 2.

Comments on Program 2.2

Statement number	Comment
1.05	Program label
1.1	Initialization statements. The sum and index J must be set to zero before the program starts because they appear on the right hand side of SET statements.
1.2	This IF statement excludes negative values of X from the sum.
1.3	This statement adds each value of X to the sum.

Comments on Program 2.2 (continued)

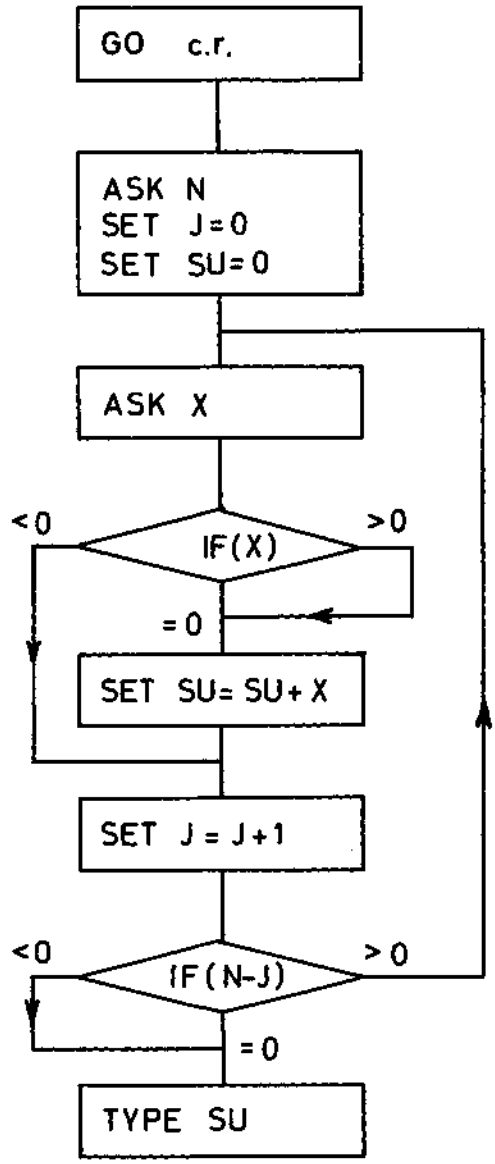
Statement number	Comment
1.4	The index J is advanced. J keeps track of how many numbers have been asked for.
1.5	This IF statement determines if the computation is finished. When N = J, N-J will equal zero and the sum will be typed out by statement 1.6. Note that J can never be more than N but that the logic of the IF statement demands a statement number for that contingency.
1.6	This statement types out the sum, SU.

The GOTO Statement

The GOTO statement has the form

```
GOTO n
```

where n is the number of a statement in the program. It unconditionally transfers control to that statement.

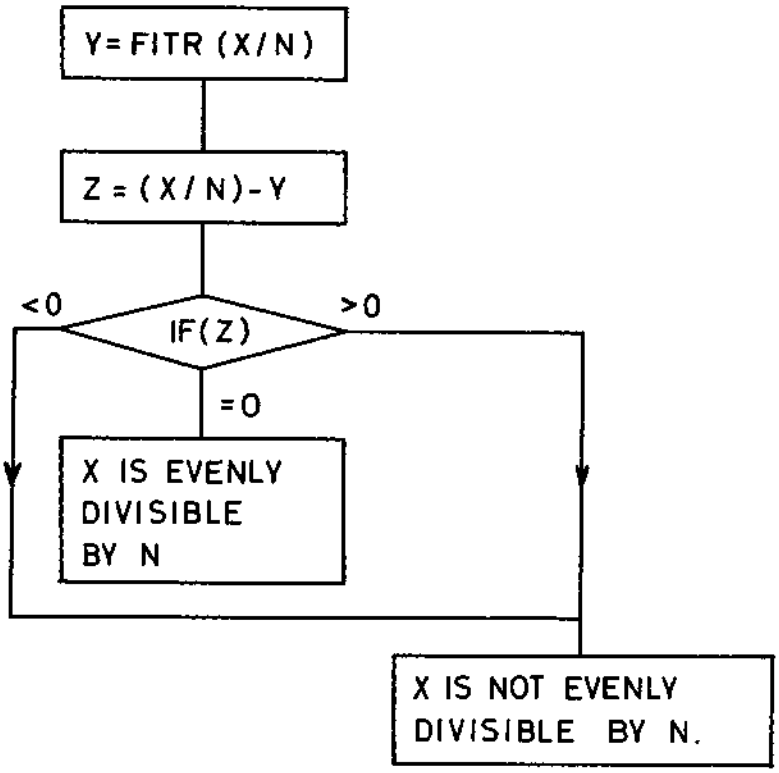


Flow Diagram 2.

Exercise Set 3.

Pencil and Paper

1. In FOCAL the function FITR (X) designates an integer equal to the integer part of X. Thus FITR (10.7) = 10. Note that FITR does not "round off". This function may be used to detect whether or not a number, X, is divisible by another number, N, an integral number of times or not as follows. (See flow diagram 3.)



Flow Diagram 3.

Write a FOCAL program to implement this flow diagram. Have it print out either: X IS DIVISIBLE BY N or X IS NOT DIVISIBLE BY N

2. Write a program to count and sum a set of numbers input with an ASK statement.

Define a particular number which will indicate the termination of the set to be summed and which, when entered in the input, will not be included in the sum. Begin by drawing a flow diagram for the program.

3. Predict the output of the following program:

```
Ø1.Ø5 C EXERCISE SET 3 QUESTION 3
Ø1.1Ø SET X = 1;  SET Y = 2
Ø1.2Ø IF (X-Y) 1.3,1.4,1.6
Ø1.3Ø TYPE "A"
Ø1.4Ø TYPE "B"
Ø1.5Ø TYPE "C"; SET X = X+1;  GOTO 1.2
Ø1.6Ø Q
```

(Take a guess at what the command in 1.6 means.)

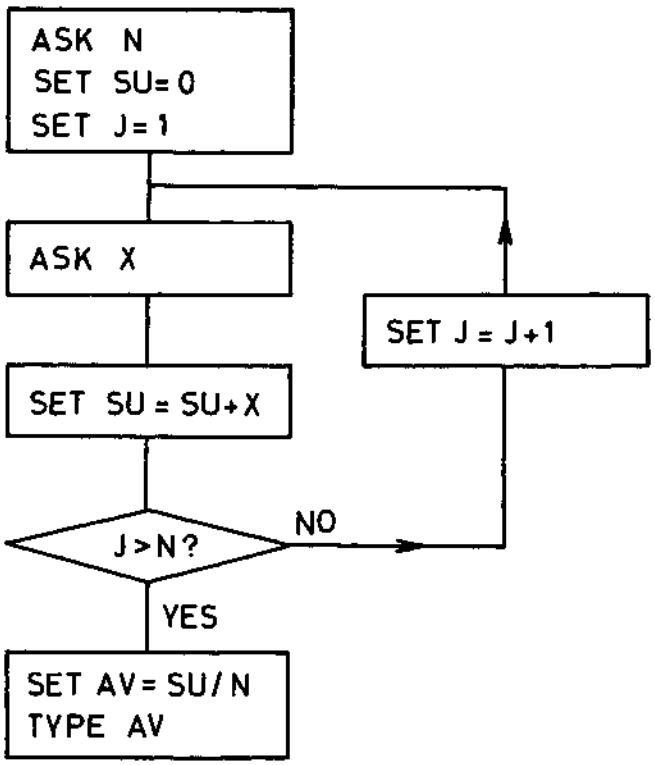
Computer Exercises

4. Load the program of pencil and paper exercise 3 above to verify your prediction.

5. Load and test your programs for exercises 1 and 2 above.

6. Write a program to determine if any particular four numbers X, Y, Z, and N to be supplied as input obey the relation X↑N + Y↑N = Z↑N. Have the computer answer with a simple yes or no. Please do not write your answer in the margin of this page.

THE AVERAGING ROUTINE



Flow Diagram 3.5

The FOR Statement

Often we perform certain calculations a specified number of times. For example, a routine to average N numbers might be

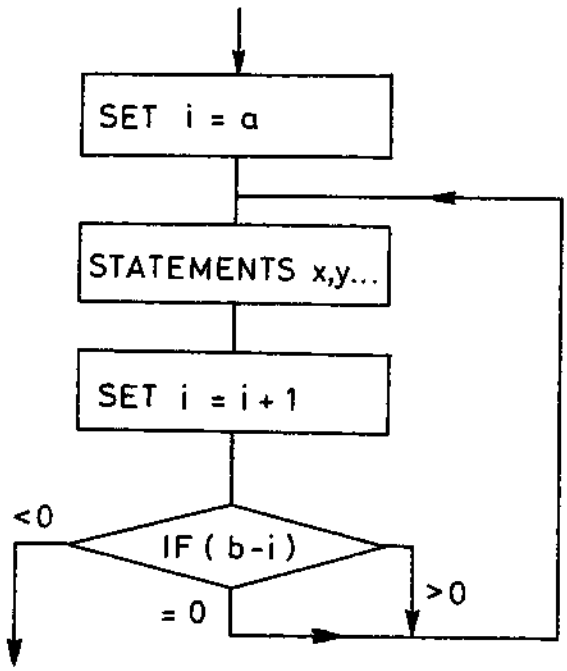
```
Ø1.Ø5 C PROGRAM EXAMPLE 2.3, AVERAGING ROUTINE
Ø1.1Ø ASK N,!;  SET SU = Ø;  SET J = 1
Ø1.2Ø ASK X,!;  SET SU = SU+X
Ø1.3Ø I  (N-J) 1.4;      SET J = J+1;  GOTO 1.2
Ø1.4Ø SET AV = SUM/N;  T % 8.Ø4,AV,!
```

This program is diagrammed in Flow chart 3.5.

The operation wherein a set of calculations must be performed again and again while an index advances from one limit to another (e.g., 1 to N) occurs so often in computing that all algebraic languages have a compound statement for it. FOCAL uses the FOR statement; its usual form is

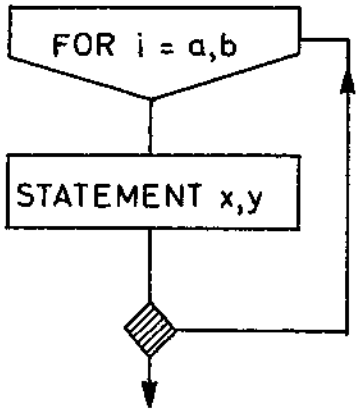
```
FOR i = a,b;  statement x,y...
```

where i is an index which is initiated at a and advanced till it exceeds b. At each increment including i = a and i = b, and all steps in section statements x, y,.. are executed. When i > b control passes to the next statement and the statements x, y, are not executed. The flow diagram for a FOR statement is:



Flow Diagram 4.

We shall adopt the following notation for a FOR statement:



Flow Diagram 5.

The averaging routine of 2.3 may be written:

```
Ø1.Ø5 C PROGRAM EXAMPLE 2.4, ILLUSTRATING FOR LOOP
Ø1.1Ø ASK N,!;S  SU=Ø
Ø1.2Ø FOR J = 1,N;  ASK X,!;   SET SU = SU+X
Ø1.3Ø SET AV = SU/N;  TYPE %8.Ø4,AV,!
```

Note that the ASK and SET commands of 1.2 are executed for each incrementation of J.

Non-Unitary Incrementation of FOR Statements

Another form of the FOR statement is

```
FOR  i = a,del,b;   statement x, y...
```

The interposed del will be the increment in the statement. It may be either integer or fractional and should be positive.* For example, the statement:

```
FOR J = 1,  .5, 3;  TYPE %3.Ø1, J,!
```

will cause the numbers

```
1.0
1.5
2.0
2.5
3.0
```

to be printed out.

*FOR statements will accept negative increments but will not terminate unless their index exceeds b.

Both the bounds and the increment a, b and del may be number variables or algebraic expressions.

The DO Command

This command gives FOCAL true subroutine capability. It allows the normal sequence of command execution to be interrupted and control to be transferred to another subsection of the program before returning to the main program.

Its form is:

```
DO  x
```

where x is a statement number like 1.15 or a block number like 3.0. In each case DO x will cause either statement x, or all the statements in block x to be executed, provided execution is not interrupted by another logical command. The use of the DO statement may be seen from the following example.

Suppose we wish to calculate the number of different linear arrangements (permutations) of base pairs in a nucleic acid molecule. We may do this with the equation for the permutations

$$N=M!/(B1!\cdot B2!\cdot B3!\cdot B4!\cdot B5!)$$

where N is the number of permutations, M is the total number of base pairs and B1 is the number of base pairs of type 1, etc. To perform this calculation we must obviously compute a number of factorials. (Recall that z! of a positive integer z is the product 1x2x3x...z.) The following program uses a DO loop and subscripted variables to compute the permutations. Note that B_1 is indicated by using parentheses to designate the subscript.

```
Ø1.Ø5 C PROGRAM EXAMPLE 2.5 ILLUSTRATING DO COMMAND
Ø1.Ø6 SET PR = 1;  S  M = Ø
Ø1.1Ø ASK K,!;  FOR J = 1,K;  ASK B(J);  S M = M+B(J)
Ø1.2Ø SET X=M;  DO 6.Ø;  SET NU = Y
Ø1.3Ø FOR  J = 1,K;  SET X = B(J);  DO 6.Ø;  SET PR = PR*Y
Ø1.4Ø TYPE NU/PR
Ø1.5Ø QUIT
Ø6.1Ø SET Y = 1;  IF (X) 6.4,6.3,6.2
Ø6.2Ø FOR L = 1,X;  SET Y = Y*L
Ø6.3Ø RETURN
Ø6.4Ø TYPE "FACTORIAL OF NEG. NUMBER REQUIRED"; QUIT
```

Termination of a DO Statement and Passage of Control outside of a DO Statement Range

We will define the range of a DO statement as either 1 line as in DO 3.24 or an entire block as in DO 5.0. In any event, control will be returned to the statement immediately following the DO statement (which, in a compound command, may be on the same line as the DO statement, e.g., see statement 1.30 of example 2.5, above) if one of several conditions is met. These are:

A) The statement or block has been completely executed.
B) Control has been given back to the following statement due to the program encountering a RETURN statement.
C) An IF or GOTO has transferred control to a statement outside of the range of the DO, in which case the one single statement is executed (provided no further IF or GOTO is encountered) and control is returned to the statement following the DO.

Warning: This last exception often gives FOCAL programmers trouble. The following section discusses the sort of trouble it can cause and the way to get out of it.

Breaking Out of a FOR Loop Which Calls a DO Statement

Often a programmer may wish to leave a FOR loop before it has reached its upper limit. To do this one may be tempted to exit via an IF statement, but he will be foiled in this attempt by virtue of exception C) noted above if his FOR loop has employed a DO statement. This sounds terribly complicated, because it is, really; but complicated or not, it often occurs in actual programming practice.

Look at the following program which attempts to find the smallest integer number which evenly divides a supplied number, X.

```
1.Ø5 C PROGRAM 2.6
1.1Ø ASK X,!
1.2Ø FOR J = 1,1ØØ;  DO 4.Ø
1.3Ø TYPE " THE SMALLEST INTEGRAL FACTOR IS ", J
1.4Ø QUIT

4.1Ø SET Y = FITR(X/J)*J
4.2Ø IF (X-Y) 4.3, 1.3, 4.3
4.3Ø RETURN
```

If the number X is evenly divisible by J then Y will be equal to X, and statement 4.2 will transfer control to statement 1.3. But because of exception C) noted above, the program will not go on to execute statement 1.4 and quit, but rather will return to the DO loop and continue to find and print out values of J which evenly divide X right up to J = 100. A way to avoid this problem is given in program 2.7

```
1.Ø5 C PROGRAM 2.7
1.1Ø ASK X,!
1.2Ø FOR J = 1,1ØØ;  DO 4.Ø
1.3Ø QUIT

4.1Ø SET Y = FITR (X/J)*J
4.2Ø IF (X-Y) 4.3,4.25,4.3
4.25 TYPE "THE SMALLEST INTEGRAL FACTOR IS",J; SET J = 1Ø1; RETURN
4.3Ø RETURN
```

Comments on Program 2.5

Statement Number	Comment
1.06	This initializes a product variable, PR at 1.
1.1	This statement asks for M, K, and then (in a FOR loop) K values of B, i.e., B(1), B(2)...B(k). Note that in a subscripted variable the subscripts appear in parentheses immediately after the variable name.
1.2	To compute the numerator the argument of a subroutine 6.0 is set equal to M. Then all the statements starting with 6 are executed in order by the DO 6.0 command. Lastly, the numerator is set equal to Y, the output variable of the subroutine. The reader should verify that this routine computes Y=X!
1.3	Factorials of all B's are then taken in a FOR loop, and the product, which is the denominator, is formed.
1.4	The result is typed.
1.5	This command halts execution and prevents statements in 6.0 from being executed again.
6.1	This statement initializes the output variable Y (which will be used as a product term). The IF statement rejects negative arguments, sets the output, Y = 1 for zero arguments and passes control to ordinary factorial computation for positive values of X.
6.2	The value of X! is computed in a FOR loop.
6.3	This statement causes return to the main program.
6.4	This statement is reached from a negative argument to the factorial routine in 6.1 and causes a warning to be printed and execution to cease.

Program 2.7 avoids the trouble encountered by 2.6 by actually changing the value of the index of the FOR loop, making it exceed its upper bound, and thus forcing the program to leave the FOR loop before it makes the next iteration.

This is a perfectly valid way to leave the FOR loop, and in some cases it may be the only way to get out of it at the time desired.

Miscellaneous Statements: QUIT, RETURN, CONTINUE

QUIT halts program execution whenever it is encountered. It is often used to halt the program when an error is encountered, or to prevent execution of a subroutine at the end of a program. It is not necessary to use a QUIT statement to terminate a program; it will always stop when it gets to the last statement.

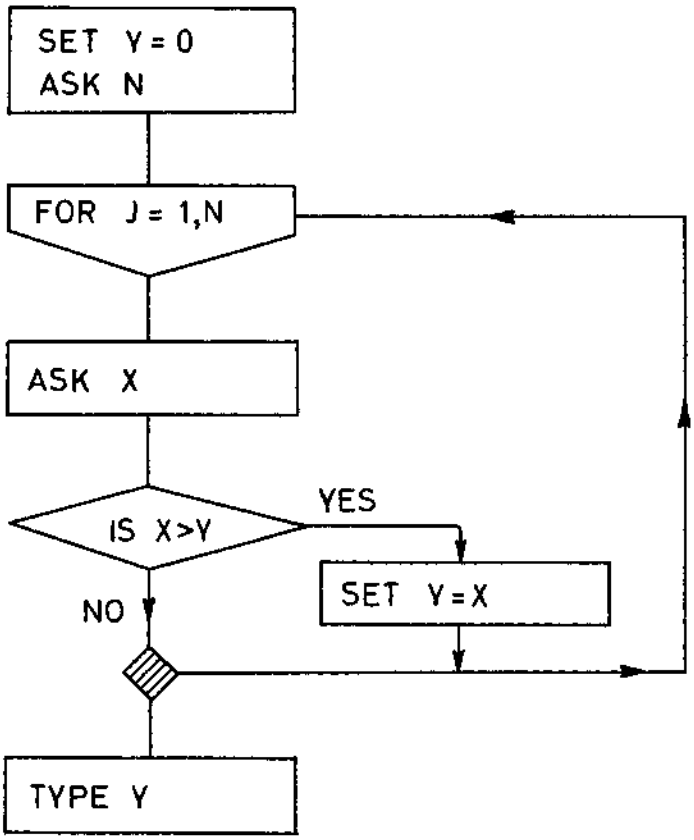
RETURN is used to halt the execution of a block in a DO statement. It returns control to the statement following the DO which initiated the block.

CONTINUE is indistinguishable from a Comment statement (it has the same abbreviation). It merely provides a reference point for a GOTO or an IF. Its use may prevent extensive changes in a program when a statement which was formerly the direction of an IF or GOTO is eliminated. When a CONTINUE statement is encountered, control passes to the statement with the next highest statement number.

Exercise Set 4.

Pencil and Paper

1. The following routine depicted in Flow Diagram 6 inspects a list of N numbers and sets an output variable, Y, equal to one of them. If N = 5 and the list of x's were -1,3,7,2,1; what would Y be after the program had executed?



Flow Diagram 6.

2. Draw a flow diagram for the following program:

```
Ø1.Ø4 C EXERCISE SET 4 QUESTION 2
Ø1.1Ø ASK N,!;  F  J=1,N;  ASK X(J),!
```

```
Ø1.15 S  SU=Ø;  S S2=Ø
Ø1.2Ø F  J = 1,N;   D 4.Ø
Ø1.25 S  AV=SU/N
Ø1.3Ø TYPE SU,S2, AV,!;Q

Ø4.1Ø S SU=SU+X(J); SET S2 = S2+X(J)↑2
```

3. An often used algorithm for ordering a list of numbers from smallest to largest is to start with the first number in the unordered list and compare it with every other number in the list. If one is found which is smaller than the first, the two are exchanged, and the smaller number becomes the "test" number with which all others are compared. Every time a number smaller than the "test" number is found, it becomes the test number and the former test number is put in its place. After the entire list is scanned in this fashion, the most recent number is placed first in the list, the second number in the list becomes the test number and the remaining numbers in the list (excluding the first number) are searched again. This entire procedure is repeated for all the numbers in the list, and when it is finished the list is ordered. A FOCAL program for the entire sort routine is:

```
Ø1.Ø4 C EXERCISE SET 4 QUESTION 5
Ø1.15 ASK N,!;  FOR J = 1,N; ASK X(J),!
Ø1.2Ø FOR J = 1,N-1;  FOR K=J,N;  D 3.Ø
Ø1.3Ø FOR J = 1,N;  TYPE X(J),!

Ø3.1Ø IF (X(J)-X(K))3.5,3.2,3.2
Ø3.2Ø SET Y = X(J)
Ø3.3Ø SET X(J) = X(K)
Ø3.4Ø SET X(K) = Y
Ø3.5Ø RETURN
```

Draw a flow diagram for this program.

Computer

4. Write a program for the pencil and paper exercise 1. above and run it on the computer.

5. Type the program in exercise 3 into the computer and test it for a string of N=5 numbers. How does the execution time vary with N? Use a watch to time it.

6. Everyone knows that the positions of a set of on-off switches may be coded into a binary number. For example, the binary number corresponding to the starting address of FOCAL is 000010000000 because the 5th switch on the switch register is up. It is also obvious that three binary numbers may be used to code any number from 0 to 7. Thus, 111 is 7, 010 is 2 and 011 is 3. What may be new to the reader is the trick of dividing a binary number up into groups of three to translate it to an octal number. (An octal number uses the base 8 and has only the digits 0,1,2,3,4,5,6,7. Thus the octal number 12 is 1·8 plus 2 or 10 in decimal. Octal numbers are often written

with a subscript. Thus 23_8 is $2 \cdot 8+3$ or 19_{10}.) To return to the conversion of binary to octal: the binary number 000010000 may be written in groups of three's as

000 010 000 000 and translates into octal as

0 2 0 0 or 200_8.

Thus the binary number 101111110010 becomes

101 111 110 010 5 7 6 2 or 5762_8.

Octal numbers are often used in computing. You are asked to write a program to convert octal into decimal. Given an octal number X, your program should return a decimal number, Y. This problem is not a trivial one and will probably require some thought. Your program need not handle numbers larger than 7777 in octal.

CHAPTER IV

SUBSCRIPTED ARRAYS

Arrays

It has already been mentioned above that FOCAL permits variables to be subscripted, with the subscript or subscript expression following the variable in parentheses. Thus:

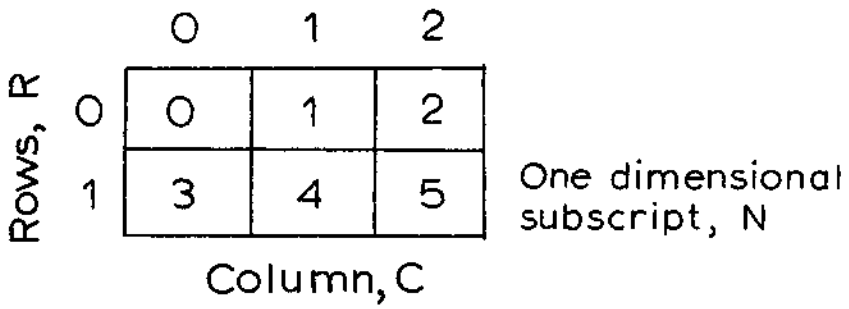
```
X(1)
X(J+2/N)
VA(R+RM*C)
```

are valid subscripted variable names. Note that in FOCAL no "dimensioning" statement is required; FOCAL recognizes that a variable is subscripted when it sees it and saves enough space in memory for every subscripted variable mentioned.

In FOCAL, zero is a valid subscript, and it is conventional to start arrays with zero as the beginning subscript for reasons which will soon become apparent.

Multidimensional Arrays

FOCAL allows only a one dimensional subscript, but two or more dimensional arrays are easily folded into a one dimensional array as follows:



We simply number the elements of the array starting with the upper left hand corner and proceed left to right varying the column first and then the row.

Let us call the total number of rows RM and the total number of columns CM. In

our example RM = 2 and CM = 3. The one dimensional subscript, N, is given by the following function of R, the row index, C, the column index, and RM, the total number of rows. One may verify by inspection that:

$$N = R \cdot CM + C$$

We will use this formula in the following program in order to show how we may store a two by three matrix as a one dimensional array (often called a vector). The program will accept the matrix and print out the values of the matrix in linear order together with their subscripts.

```
01.10 C PROGRAM 4.1
01.15 A ?RM,CM?,!
01.16 T %2,  "NOW ENTER THE ARRAY OF ", RM, " ROWS AND", CM,"COLUMNS",!
01.20 F R=0, RM-1; T !; F C=0,CM-1; A A(R*CM+C)
01.25 T ! !
01.30 S M=CM*RM-1
01.35 T "SUBSCRIPT   VALUE",!
01.40 F J=0,M; T %5,J,%11.04, A(J),!
*GO
RM,2 CM3
NOW ENTER THE ARRAY OF   2 ROWS AND   3 COLUMNS

4 2 7                                  (User entered this.)
3 5 8

SUBSCRIPT      VALUE
     0         4.0000
     1         2.0000
     2         7.0000                  (Computer supplied this.)
     3         3.0000
     4         5.0000
     5         8.0000
*
```

Comments on Program 4.1

Statement Number	Comment
1.15	This statement asks for the number of rows and columns in the matrix.
1.16	This statement gives an instruction to the user.
1.20	This statement uses two nested FOR commands to ask for the elements in the matrix. Note that the ! causes a carriage return after each row has been entered.

Comments on Program 4.1 (continued):

Statement Number	Comment
1.30	This statement computes M the highest subscript of a matrix element.
1.40	This statement prints the values of the matrix together with their subscripts.

One very practical use for such a program as that of 4.1 concerns the handling of data which is to be processed by another program and which must be entered into the computer absolutely correctly. It is simply true that if the ordinary mortal is asked to type more than ten numbers into the computer with no errors, he or she is in for a very bad time. Consequently it is often practical to employ a matrix "editor", a program which asks for the numbers, permits the user to make corrections, and stores them for later printing or punching on paper tape. Such a program which has found use in our physiological research laboratory is the following:

```
Ø1.Ø5 C PROGRAM 4.2
Ø1.1Ø C MATRIX EDITOR
Ø1.2Ø A ?CM,RM?,!
Ø1.3Ø F R=Ø, RM-1; T !; F C=Ø,CM-1; A X(R*CM+C)
Ø1.4Ø Q

Ø2.1Ø D 5.Ø
Ø2.2Ø F R=Ø,RM-1; T !;F C=Ø,CM-1;T %8.Ø4,X(R*CM+C)
Ø2.3Ø D 5.Ø;Q

Ø5.1Ø T !,"                    "
*
```

Just as in program 4.1, this program asks the user to enter the matrix. But the values of the matrix are not printed out until the user gives the additional command, GOTO 2.1.

The user need not give this command until he has made sure that all the numbers have been entered correctly, or he may have the computer print out the list as many times as he wishes by giving the command repeatedly.

Errors in the matrix may be corrected by giving direct commands to set erroneous values to their correct ones. In the following output from the program an error was made in entering the third element in the first row (subscript=2). That element is corrected and the matrix is printed out correctly. If desired, a paper tape could be punched (with leading spaces) at the same time by simply turning on the tape punch on the teletype while giving the command GOTO 2.1.

```
 C OUTPUT FROM 4.2
*GO
CM,3 RM2

1 2 4
4 5 6 * S X(2)=3                    User types this.
*G 2.1

   1.0000      2.0000      3.0000
   4.0000      5.0000      6.0000                Computer types this.
                                  *
```

Note that the program stopped after the matrix was read in at statement 1.4 (indicated by the asterisk after the 6), and that the user corrected the third element of the first row of the matrix (X(2)) to be 3 rather than 4 by resetting the value with a SET statement. The computer was then commanded to print out the matrix with the command G 2.1.

Exercise Set 5.

Paper and Pencil

1. Extend the notation for a two dimensional array to encompass three dimensions. Call the elements of the third dimension, "planes". Let the number of planes be PM and write an algorithm for the one dimensional subscript of a matrix of RM rows, CM columns and PM planes.

2. Write a program to read in a two dimensional array and to compute the sum of all the elements in every row and every column. Have the computer program check itself by adding up the sum of all the column totals and comparing it with the sum of all the row totals. Both numbers should be identical with the sum of all the elements in the array. Have the computer write TILT if it is not.

3. Write a program which reads in a m by n matrix and finds the row and column number of the maximum element in the matrix.

4. A three dimensional array has three rows, three columns and three planes. This might be visualized as a cube with the elements 0 and 26 at opposite corners; what other elements are also at opposite corners of the cube?

Computer

5. The following program performs the operation of matrix multiplication. (For a definition of matrix multiplication, see any elementary finite mathematics book, e.g. page 244, Kemeny, Snell and Thompson's Introduction to Finite Mathematics, 2nd edition, Prentice Hall.)

```
Ø1.Ø5 C PROGRAM 4.3 MATRIX MULTIPLICATION
Ø1.1Ø A ?RM, CM, KM?,!
Ø1.2Ø F R=Ø,RM-1; T !; F C=Ø,CM-1; A A(R*CM+C)
Ø1.25 T !; F R=Ø,CM-1;T !;F C=Ø,KM-1;A B(R*KM+C)
Ø1.3Ø D 4.Ø; F R=Ø,RM-1;T !;F C=Ø,KM-1;T %8.Ø4,P(R*KM+C)

Ø4.1Ø F R=Ø,RM-1; F C=Ø,KM-1; D 6.Ø

Ø6.1Ø S T=Ø; F L=Ø,CM-1;S T=T+A(R*CM+L)*B(L*KM+C)
Ø6.2Ø S P(R*KM+C)=T
*
```

The program proports to multiply a matrix of RM rows and CM columns by one of CM rows and KM columns. Check the program against the definition of matrix multiplication to see if, in theory, it should work. Then enter the program into the computer and try it for several examples of matrix multiplication given in a textbook. Determine in this fashion if the program performs correctly.

6. Execute and debug the programs of 2 and 3 above.

7. Write a bookkeeping program which accepts debits and credits from any one of 20 accounts (numbered from 1 to 20) and which supplies debit and credit totals for each account on demand. Each item will be entered as input by first giving the account number, the letters DR or CR and the amount. The program will supply account totals when account number -1 is entered. (Note: when one responds to an ask statement by giving letters instead of numbers, these letters are assigned numerical values by FOCAL. To find out what number is assigned to a letter or group of letters use the following statement:

```
1.1 ASK X; TYPE %8,X,!
```

When the program asks for X, respond with a letter or letters and see what value it types out. This treatment of letters in the input is an idiosyncracy of FOCAL.)

CHAPTER V

PROGRAMMING HINTS FOR INTERACTIVE COMPUTING ON MINI COMPUTERS: TESTING AND DE-BUGGING

Documentation

A central problem in all computing is adequate documentation of the program. Perhaps the best way to see the problem is to imagine that you have been given a program to perform a computation which you very much want to do. The program has been written by someone else and you are trying to understand how to use it. What questions would you have about the program? Some of them might be:

1. Does the program really work? Has the author tested it, and if so, how?

2. How do I enter the input data? Do I know what quantities the input and output variables stand for?

3. By what algorithm(s) does the program work? Is the accuracy sufficient for this task?

4. If I desire to modify the program for my own needs, can I figure out how the program works in order to do this?

5. Are there any options or restrictions regarding the input data that I should know before I try to use the program?

All of these questions and many like them should be answered by the documentation of the program. By documentation is meant descriptive comments embedded in the program and any supplementary descriptive material about the program to which the normal user would have access.

By now you may be saying, "Surely this does not concern me, as I only write programs for my private use." But the sad facts are that it does concern you, because, like all other mortals, if you write a program and do not look at it for a few months, you may not even recognize it as your own, let alone understand what it was all about. Hence, even if you are only writing notes to yourself, documentation is an important factor in your efficiency as a computer programmer.

Using Comments in Mini Computer Programs

One problem unique to programming on a mini computer is the limitation of memory space. This severely restricts the number of comments that can be added to a program and stored with it. One way to get around this problem is to put all of the comments concerning the program's documentation in one section on consecutive lines. Then, if user space is at a premium, these comments, or the entire section can be erased before the program is run, thus freeing up needed space for program variables.

It should be noted that in many situations, notably batch processed languages, comment cards are essentially ignored by the computer and hence can be used very liberally throughout a program. In general, documentation of this sort which is a permanent part of the program is to be much preferred over any scheme of ancillary papers, e.g., keeping descriptions of programs in notebooks, etc.

Essential Information

To the author who has just written a program the program itself is its own documentation. But as time goes by, the recording of some information becomes more and more important. In complicated mathematical work, perhaps the most important item is the algorithm or method the computer uses to solve a problem. This information might well be noted in a comment of the sort

```
1.Ø5 C NEWTON'S METHOD P. 35, CONTE, ELEM. NUM. ANALYSIS. MCGRAW HILL '65
```

If a program has been revised several times and is one of many programs used on a project, then a Name and a Date (to indicate which version it is) become essential descriptive data. Such a comment might be

```
1.Ø4 C NON LINEAR EQUATION SOLVER, 4 APRIL 1972
```

If a number of programs are used to process data in a project, it is often important to have printed in the program output which program's output it is. Thus the output section of a program might begin with the statement:

```
5.1Ø T "NLQS 4 APR '72",'
```

Other Information

If one were making a library of programs for the use of other persons a great deal more information about each program would be useful. We have already mentioned:

Program name
Date (to identify version)
Algorithm source

In addition one may wish to supply the following information:

Variable definitions, i.e. what they stand for
Input data required
Output data printed
Restrictions on input data
Options available to user
Program logic, and what major sections of program do
Applications, i.e., what the program is good for.

Such information about programs may be conveniently filed and made accessible to users, preferably in a form that they can take with them to the teletype console and use along with the actual program.

Debugging Programs

Errors in programs are called "bugs" and getting rid of them is called "debugging". It is rare that a programmer writes more than a ten line program and finds that it works the first time. Usually the errors in a program are obvious "typos" which, in interactive computing can be corrected at once. Sometimes, however, the problem lies in faulty logic on the part of the programmer, and considerable effort may be required to find the error.

Invisible Errors

Some errors are "invisible". The programmer can be looking right at them and still not see them. A typical "invisible" error for a beginning programmer is the absence of the multiplication sign, e.g.

```
1.Ø5 SET KE= .5MV↑2
```

which should be

```
1.Ø5 SET KE= .5*M*V↑2
```

Because the programmer has always omitted multiplication signs in ordinary mathematics, it is hard to learn to put them in programs, and hard also to find the error. Perhaps the fastest way to find invisible errors is to have someone else look at the program. This will often relieve an incipient temper tantrum born out of sheer frustration. The beginner may take heart that almost all "invisible errors" eventually become very visible to the experienced programmer.

Errors in Logic and the Method of Inserted TYPE Statements

The more powerful a command is, the more potentiality it holds for error. Thus the FOR statement and the IF statement are often trouble makers in this regard, and last but not least, subscripts (which never seem to have the values they ought to have) often lead one to difficulty. Most of these problems can be solved by the following method:

1. Isolate the trouble spot by inserting TYPE statements to see how far the program has gotten. For example, if your program contains two statements, 2.30 and 2.40 and if you wish to see if it has gotten beyond 2.30, you need only insert the statement 2.35 TYPE "HERE",! (or any other message you desire) and, if the computer prints it out, the program must have gotten as far as 2.30. Since FOCAL usually identifies the statement which is causing it trouble, you will not have to do this very often. Sometimes, however, the FOCAL error code refers to a statement which called another section of the program with a DO statement. In this case, while the offending statement may be contained in the section called by the DO, the error code may refer you to the calling statement. In this event, this isolation technique may be valuable.
2. If an error occurs in a long algebraic expression, you may wish to isolate components of that expression, assign them variable names and then print out the values of these intermediate variables with a TYPE statement. Once

you find the trouble you can rewrite your original expression and get rid of your debugging "scaffolding". Be particularly suspicious of the values of subscripts as well as the values of the arguments of IF statements.

3. If a particular section of your program is producing erroneous results, but you can't see why, try breaking that section into smaller subsections and test each one individually with data which you supply with direct commands. Once you are convinced the parts are working, try all the sections together, again with simplified input data which you know how the program should handle. After this works, then go back to the original problem and try it again.

4. If your program is still giving you trouble despite these techniques, then put it aside for a while and do something else. Come back to it later when you are fresh.

5. When all else fails, seek human help. Try to find someone who is paid for such services, like a teacher or a consultant, rather than a harried fellow student.

An Additional Debugging Aid

In FOCAL the statement

```
TYPE $
```

will cause the printing out of all the internally stored variables in a FOCAL program. The statement must appear on a line by itself but it may appear in an indirect statement. For example, the following program:

```
Ø1.1Ø A X,Y,!
Ø1.2Ø S Z = X*Y↑3
Ø1.3Ø T Z, !
Ø1.4Ø T $
```

when executed will yield this output:

```
*

 G
:2 :3
= Ø.54ØØØØ+Ø2
X@(ØØ)= Ø.2ØØØØØE+Ø1
Y@(ØØ)= Ø.3ØØØØØE+Ø1
Z@(ØØ)= Ø.54ØØØØE+Ø2
*
```

Note that FOCAL invents a second symbol for a one letter variable.

The TYPE $ statement may be used as many times as is necessary in a program to investigate the state of all stored variables.

Testing Programs

Simply because a program produces plausible output one should not infer that it produces correct output. Every program should be tested in some fashion before it is accepted as correct. In statistical programs, for example, it is often convenient to try the program out on sample data supplied from the statistics book from whence the algorithm came.

In complicated programs it is often convenient to test the parts of the program separately, supplying input data to the parts with direct commands and extracting their output in the same way.

Often it is practical to test a program on simple data for which we know the answer and to infer, if permissible, that if the program can handle such simple problems correctly, it can also do the more complex ones. For example, if a matrix multiplication program can multiply two small matrices of different numbers of rows and columns, it could probably handle a much larger job as well (provided of course that it didn't run out of variable space).

Hints for Programmers

It is obviously easier to document and debug some programs than others. These facts suggest certain guidelines which, if followed, will make the programmer's job easier. I have gathered these together in the following list of "hints" which I hope will make your job easier.

1. In any interactive language, leave space between lines so that additional statements can be inserted. In FOCAL it is good practice only to use the first decimal place, i.e. 3.1, 3.2, for line numbers in the initial writing. This will allow the easy insertion of additional material or debugging aids.

2. Leave the first section (1.0) of a FOCAL program blank, or use it only for documentation. This will permit the easy addition of an "outer" program which may call your original program several times or use it in some other way you had not originally envisioned.

3. Break the separate tasks of your program into separate units and assign a separate section number to each unit. For example, it may be necessary to compute factorials in a program. Use a separate section for this portion of the program. If you do, you will have a factorial subroutine which can be used in other programs, as well as providing yourself with a section of the program which is easy to debug.

4. Use plausible names for variables. For example, I find it easy to remember that SU stands for sum, SD for standard deviation, etc. It is easier to remember these names than A,B,C etc. You may also wish to get into the habit of using the FORTRAN convention to distinguish integer variables from non-integer ones. FORTRAN reserves the initial variable letters I-N for integer variables. I have found this a useful way of reminding myself that a variable is supposed to have integer values. (And it saves me trouble when I write FORTRAN programs.)

5. Avoid long algebraic expressions. I find these hard to debug, and hence I habitually write several statements in place of a long single one. For example, rather than

```
3.1 SET X =[(-B+FSQT(B↑2 -4*A*C)]/  (2*A) +[CF*FLOG(Y↑3)]
```

I tend to write:

```
3.1 SET X = -B + FSQT(B↑2-4*A*C)
3.2 SET X = X/(2*A)
3.3 SET X = X + CF*FLOG(Y↑3)
```

Of course, it must be admitted that this is wasteful of machine time and space, but since it saves my time, I favor this simple, step by step method. If needs be, once you have the program de-bugged you can recompress these statements into their tighter form.

6. Avoid tricky logic, even if it works. It is obviously possible to write a clever program which dazzles the user and confounds the persons who desire to find out how it works. A few months after you write the program you will find that you now belong to that latter group--so who were you trying to impress?

7. Don't waste too much time on formatting output. Output should be clearly labled, but it needn't look like it was set by a printer. Format your output as much as comes naturally to you--your ability in this area will naturally increase, without your engaging in great typographical endeavors.

8. Avoid the necessity of reinvention of your programs; document them as you write them.

Building a Library of Programs

Many routines, if written in a sufficiently general form and well documented will be useful for years to come. These are easy to save on paper tape which may be mounted, together with a print out of the program on the same page in a small evelope. A notebook of such routines is a valuable tool for any scientist.

It is good practice to acknowledge authorship of routines that you use in your published work if you did not write them yourself.

Exercise Set 6.

Pencil and Paper

1. Document the matrix multiplier of Exercise Set 4. Add comment statements to the program itself and supply a separate "Program Description" with sections on all the "Other Information" suggested in the text.

2. Document the bookkeeping program you wrote in Exercise 7 of Exercise Set 5 above.

Computer

1. Obtain a documented FOCAL program written by anyone but yourself. Run the program and convince yourself you can use it as it was intended. Write a critique of the program's documentation, distributing praise or blame as warranted, along with your constructive suggestions for its improvement.

CHAPTER VI

BASIC FOR USERS OF FOCAL

While FOCAL is a powerful and elegant language, and the only interactive algebraic computing language which will run on a 4000 12 bit word machine, it is nonetheless not the most widely used interactive computing language in the country. That language is BASIC which was written at Dartmouth College by John Kemeny, Thomas Kurtz and a host of undergraduate students. Since the writing of BASIC, it has become a popular interactive language which is now implemented on many machines. Because BASIC is very similar to FOCAL, and because the user is likely to encounter it in his computing career, a chapter on BASIC is included here.

Since it is assumed that the reader is already familiar with FOCAL we shall consider the essential features of BASIC in a rather rapid fire sequence.

Variable Names

Variable names in BASIC are similar to those in FOCAL with the exception that the second character of a BASIC variable name, if there is one, must be a number. Thus SU is a legal variable name in FOCAL and illegal in BASIC.

Statement Numbers

Unlike FOCAL, BASIC has no decimal points in its statement numbers, nor does it have its statement numbers in sectional blocks like 7.0. All statements in BASIC are numbered in integers between 0 and 9999.

Order of Execution

BASIC executes statements in statement number order unless the sequence is altered by logical statements.

Order of Statement Entry

Like FOCAL, statements in BASIC may be entered in any order desired; they will be executed in statement number order.

Algebraic Expressions

The symbols used in BASIC algebraic expressions are identical to those in FOCAL, e.g., +, -, *, /, ↑.

Assignment Statement

SET in FOCAL becomes LET in BASIC, e.g., LET X = Y↑2.

Abbreviations

The only abbreviation allowed in all versions of BASIC is REM for REMARK which

is equivalent to C for COMMENT. Some versions of BASIC accept the first three letters of any command as its abbreviation.

Compound Statements

There are no compound statements in BASIC.

GOTO

GOTO n where n is a statement number has the same meaning in BASIC as it does in FOCAL.

Data Input

ASK in FOCAL becomes INPUT in BASIC, e.g., 20 INPUT A,B. BASIC responds by typing a ? on the teletype signalling that it is waiting for data. This corresponds to FOCAL's colon.

INPUT is not the only statement which allows one to enter data in BASIC. Another way is to store input in a DATA statement and read it with a READ statement. For example, the BASIC program

```
10 READ A,B
15 PRINT A*B
20 GOTO 10
35 DATA 1,2.5,3,4
40 DATA 2,2
50 END
```

will assign the values 1 to A and 2.5 to B the first time statement 10 is encountered. The next time it will use the values 3 and 4, and the next time 2 and 2, after which it will stop for lack of data. In the above example the output would be:

```
RUN
 2.5
 12
 4

ERROR  47 IN  10       (out of data)
```

The READ and DATA statements obviously always go together and allow BASIC to serve as a batch processed as well as an interactive language.

IF Statement

The IF statement in BASIC, as in FOCAL is a logical branch point. Unlike FOCAL, its argument is not enclosed in parentheses and is a logical expression rather than a number. For example, the BASIC statements

BASIC		FOCAL
75 IF X<Y THEN 77	are like the FOCAL	7.5 IF (X-Y)7.7; TYPE X,!
76 PRINT X	statements	7.7 R
77 RETURN		

The format of an IF statement in BASIC is always:

```
IF e THEN n
```

where e is a logical expression and n is a statement number. The logical symbols in BASIC are: <, <=, =, >, >=, <>.

Iteration

BASIC like FOCAL has an iterative FOR statement. Its form is

```
10 FOR I = 1 TO 5
11 REMARK ALL STATEMENTS BETWEEN 10 AND THE "NEXT I" WILL BE EXECUTED
12 REMARK EACH ITERATION
13 NEXT I
```

The logic of the BASIC FOR statements in some versions is slightly different than that of the FOCAL FOR. BASIC may first test to see if the upper bound has been exceeded and then, only if it has not, proceed to execute the intervening statements and increment the index. (Compare this with the FOCAL logic given on page 29.) Unfortunately, this is not standard practice in all BASICs.

BASIC, like FOCAL, permits loops within loops. For example, the BASIC program

```
10 FOR I = 1 TO 2
15 FOR J = 1 TO 3
20 LET K = (I*J)↑2
25 PRINT K
30 NEXT J
35 NEXT I
40 END
```

will cause the following output:

```
RUN
 1
 4
 9
 4
 16
 36
```

BASIC permits non-integer incrementation, e.g.,

```
FOR J = A TO B STEP C
```

in BASIC is the same as

```
FOR J=A,C,B
```

in FOCAL.

The TAB Function

The TAB function is a special function which may be used in a PRINT statement to specify in what column printing is to begin. Thus:

```
1Ø PRINT  TAB(5) K
```

will cause the machine to skip five spaces before printing the value of K. The argument of a TAB function may be an algebraic expression, thus permitting the easy writing of a "printer-plotter" program.

Subroutines

Like FOCAL, BASIC has a subroutine capability. Instead of FOCAL's Do n, BASIC has the command GOSUB n where n in BASIC is a statement number (never a section number as in FOCAL because there are no sections in BASIC). GOSUB n will transfer control to statement n and the program will proceed in statement number order until the command RETURN is encountered, at which point it will return to the next statement following the GOSUB statement which originally directed it to the subroutine. The equivalence between FOCAL and BASIC in this respect is best seen by writing the same program, first in FOCAL and then in BASIC, together with their outputs.

```
Ø1.Ø5 C FOCAL FACTORIAL PROGRAM
Ø1.2Ø F J = 1,5; D 2.Ø
Ø1.3Ø Q

Ø2.1Ø S S=1; F K = 1,J; S S=S*K
Ø2.2Ø T %6, J,S,!
*GO
     1      1
     2      2
     3      6
     4     24
     5    12Ø
*


 5 REM BASIC
 6 REM FACTORIAL PROGRAM
1Ø FOR J = 1 TO 5
2Ø GOSUB 6Ø
3Ø NEXT J
4Ø STOP
6Ø LET S = 1
65 FOR K = 1 TO J
7Ø LET S = S*K
75 NEXT K
```

```
80 PRINT J,S
85 RETURN
90 END

READY

RUN
 1            1
 2            2
 3            6
 4            24
 5           120

READY
```

Text Output

As in FOCAL, a PRINT statement may contain text. For example the program

```
10 PRINT "---"
20 LET Y = 10
30 PRINT "THE ANSWER IS", Y
40 END
```

will cause the output

```
RUN
---
THE ANSWER IS  10

READY
```

BASIC also provides for negative incrementation. FOR J = 8 TO 5 STEP -1 will cause J to assume the values: 8,7,6,5 in that order. Naturally, if the bounds are impossible the loop will be ignored and (unlike FOCAL, which always executes every loop at least once) the statements in the loop will not be executed at all.

Output

The output statement in BASIC is PRINT instead of FOCAL's TYPE. Every time a PRINT statement is encountered BASIC will start on a new line unless this is suppressed with a comma or a semicolon. For example,

```
10 FOR J = 1 TO 3
20 PRINT J
30 NEXT J
40 END
```

will cause the output

```
RUN
 1
 2
 3
```

while the same sequence with a comma after J in statement 20 will cause the output

```
1                    2                    3
```

A semicolon has the same effect, but it causes the numbers to be printed closer together on the page. There is no other formating of numbers in BASIC.

Formating Symbols

The comma and the semicolon are the only formating symbols in BASIC (with the exception of the TAB function, see above). There is no symbol equivalent to %, #, or !. In order to skip a line in BASIC one must insert a PRINT statement followed by no argument, e.g.,

```
35 PRINT
```

will cause a line-feed when encountered in the program.

Functions

BASIC like FOCAL has a set of internally defined functions. These together with their FOCAL equivalents are:

BASIC	FOCAL
SIN(X)	FSIN(X)
COS(X)	FCOS(X)
TAN(X)	not in FOCAL
ATAN(X)	FATN(X)
EXP(X)	FEXP(X)
LOG(X)	FLOG(X)
ABS(X)	FABS(X)
SQR(X)	FSQR(X)
INT(X)	FITR(X)
SGN(X)	FSGN(X)

In addition, the user in BASIC may define functions in FOCAL as follows, e.g.,

```
10 DEF FNG(X) = LOG(X)/LOG(10)
20 PRINT FNG(10)
30 END
```

will cause the value .30103 to be printed out. The new function FNG may be used in any algebraic expression with any algebraic expression for its argument. FNG is the $\log_{10}$ of its argument.

All user defined function names must begin with FN and end with a letter of the alphabet. Thus there may be a maximum of 26 user defined functions beginning with FNA and ending with FNZ.

Subscripted Variables

BASIC, like FOCAL, accepts subscripted variables. Unlike FOCAL:

1. All subscripted variables must be designated by a single letter, e.g., T(1) or A(5).

2. If there are more than 20 members in an array of a single subscripted variable the maximum size of the array must be mentioned in a dimension statement whose format is:

   DIM A(n), B(m) ...

   where A and B are variables and m and n are numbers giving the maximum size of the list. For example

   Ø5 DIM Z(1ØØ)

   saves memory space for 100 values in the array Z.

3. Two-dimensional subscripts may be used, e.g., T(I,J). If the array is greater than 10 x 10 these, too, must be dimensioned, e.g.,

   3Ø DIM T(16,18)

Error Diagnostics

Some versions of BASIC have very explicit error diagnostics like "ILLEGAL INSTRUCTION IN 20". Others employ error codes like FOCAL. (See Appendix 5.)

Spaces

BASIC is very forgiving about spaces. It completely ignores their presence or absence, except, of course, when they appear as literal text between quotation marks.

STOP and END Statements

STOP in BASIC plays the same role as QUIT in FOCAL. It causes execution of the program to cease. But there is another statement in BASIC which must end every BASIC program. That is the END statement. This is part and parcel of BASIC being also a batch processed language, in which the compiler must be informed of the last statement in the program.

Corrections

In order to correct a line in BASIC one may simply retype it. To delete a

line in BASIC one simply types the number of the line with nothing following it.

Running BASIC Programs

The command in BASIC corresponding to GO in FOCAL is RUN, and it is used in the same way.

Listing BASIC Programs

The command corresponding to WRITE in BASIC is LIST. Giving the command LIST will cause a copy of the program to be printed out. Some systems permit one to specify which lines should be listed by adding line numbers after the LIST command.

Direct Commands

Some BASICs permit direct commands without line numbers, just as does FOCAL. However, not all BASICs have this feature.

Additional Features of Some Versions of BASIC

BASIC is an interactive language designed to run on very large computers; hence it can afford many luxuries which FOCAL cannot. In addition, BASICs are being written by many manufacturers, and hence BASIC, much like FORTRAN, is a rapidly evolving language. Some of these powerful features of large BASICs are discussed in this section.

Matrix Operations

Some BASICs have a number of matrix operations which FOCAL does not. These are:

1. MAT READ e.g.

```
20 MAT READ A(5,6)
```

   Here, A is a matrix of dimensions 5 by 6. If A had appeared earlier in a dimensioning statement one could simply have written:

```
20 MAT READ A
```

   The values of A must be given in a data statement.

2. Summation of matrices, e.g.,

```
30 MAT X = A + B
```

   where A and B are previously defined. The new sum matrix, X, will auto-

matically be dimensioned by the operation. The expression

```
MAT A = A + B
```

is legal.

3. The product of a matrix by a scalar is given by the command

```
MAT C = (k)*A
```

where C and A are matrices and k is any algebraic expression (which must be enclosed in parentheses).

4. The product of two matrices: MAT C = A*B
Note that MAT A=A*B is an illegal statement.

5. Copying a matrix: MAT C = A

6. The transpose of a matrix: MAT C = TRN (A)

7. The inverse of a matrix: MAT C = INV(A)

8. In addition to these commands three special matrices are defined:

ZER	matrix of all zeros
CON	matrix of all ones
IDN	identity matrix

These allow the setting up of matrices without the typing in of numbers, e.g.,

```
2Ø MAT X = ZER(2,2)
3Ø MAT Y = CON(3,3)
4Ø MAT Z = IDN(3,3)
```

String Variables and String Commands

Some BASICs allow the user to define string variables whose values are simply strings, or sequences, of symbols. For example

```
LET A$  "MARY"
```

assigns the string "MARY" to the variable A$. All string variables begin with a letter and have as a second symbol a dollar sign. The value of a string variable may be printed out in a regular PRINT statement. For example, the program

```
1Ø LET A$ "MARY"
2Ø LET S =2Ø
3Ø PRINT A$ "'S SALARY IS'", S "DOLLARS PER WEEK."
4Ø END
```

when run would cause the sentence,

```
MARY'S SALARY IS 2Ø DOLLARS PER WEEK.
```

to be printed out. (Poor Mary!)

The value of one string variable may be transferred to another with a LET statement, e.g.

```
LET A$=B$
```

and several interesting functions are defined which allow the manipulation of strings. These are:

LENGTH (a) where a is a string variable name. Its value is the length in characters of the string a.

INDEX (a,b) where a and b are strings or string variables. INDEX determines if string b is included in string a. If it is, it assumes a value equal to the position in string a where the sequence of string b starts. E.g., INDEX("HOWLAND","LAND") would have the value 4.

SUBSTR(a,b) or SUBSTR(a,b,c) where a is a string or string variable and b and c are numbers or algebraic expressions for numbers. SUBSTR(A$,5) designates a substring of A$ which begins with the 5th character of A$ and includes all the rest of it. SUBSTR(A$,5,3) designates a substring of A$ which begins with the 5th character and is 3 characters long.

STR(n) where n is a number converts the number n to a string which has the same logical meaning as the number. E.g., STR(12.3) has the value "12.3".

VAL(s) where s is a string converts a string of numbers into a number of the same logical value. E.g., VAL("12.3") is the number 12.3.

String Addition

Strings may be combined with the operation of addition. Unlike ordinary addition it is non-commutative. Thus the command

```
LET A$ = "GEO" + "RGE" forms the string "GEORGE"
```

but

```
LET A$ = "RGE" + "GEO" forms the string "RGEGEO".
```

A Comment on the Advanced Features of Basic

Strings and matrix commands are not commonly found in versions of BASIC which run on mini computers, because they take up a great deal of space. But they do give that language enormous power--more power than is ordinarily found in FORTRAN, for example. These advanced commands point the way of the future of algebraic languages. As computer memory elements become less expensive, and the writers of software catch up with the designers of hardware, we can expect to see more implementations of these commands on inexpensive systems, and still newer and

more powerful commands being invented.

The RND Function

BASIC has a function which generates pseudo-random numbers. It is RND (z) where z is a "dummy" variable or number. The value of the function is a number between zero and one, which has a uniform probability density function. Actually the numbers supplied by the RND function are part of a strictly determined sequence usually generated by multiplication and truncation. In order to avoid obtaining the same sequence each time, the RND function is used for the first time in a program; some versions of BASIC allow the user to start at a different point in the sequence with the command, RANDOMIZE.

An Overview of BASIC and FOCAL

BASIC is extremely similar in elementary structure to FOCAL. Because it lacks compound commands and abbreviations it tends to be long winded, but for that reason, easier to read. It should be a simple job to translate programs from elementary BASIC into FOCAL and vice versa, and anyone who has mastered FOCAL should be able to write BASIC programs almost immediately. You may test this proposition in the following exercises.

Exercise Set 7.

Paper and Pencil

1. Translate the following BASIC program into FOCAL (From "BASIC Programming", Kemeny and Kurtz, p. 10.)

```
10 PRINT "X",  "EXP (X)",  "LOG(X)"
20 FOR X = 1 TO 3.5 STEP .5
30 PRINT X, EXP(X), LOG(X)
40 NEXT X
99 END
```

2. Translate the following BASIC program into FOCAL (From "A Guide to BASIC Programming: A Time Sharing Language", Spencer, page 83.)

```
100 REM TRIGONOMETRIC TABLE PROGRAM
110 REM COTANGENT FUNCTION
120 DEF FNC(H) = COS(H)/SIN(H)
130 LET D = 0
140 PRINT "DEGREES", "SINE", "COSINE", "TANGENT", "COTANGENT"
150 PRINT
160 PRINT "0", "0", "INF", "1"
170 LET D = D + 1
180 REM CONVERT DEGREES TO RADIANS
190 LET R=D/57.2958
200 PRINT D, SIN(R), COS(R), TAN(R), FNC(R)
210 IF D  <  45 THEN 170
220 PRINT
230 PRINT "DEGREES", "SIN", "COSINE", "TANGENT", "COTANGENT"
240 END
```

3. If you have not done so already, substitute a FOR loop for the logic of statement 210 in the above program.

4. Translate the FOCAL program example 2.2 on page 23 above into BASIC.

Computer

5. Modify the following BASIC program (shown here with its output) to sum the rows and columns of a 4 by 4 matrix of your own choosing.

```
 10 REM MATRIX ROWS AND COLUMNS SUMMATION PROGRAM
 15 READ N
 20 FOR I = I TO N
 22 LET S1=0
 25 FOR J = 1 TO N
 30 READ A(I,J)
 35 LET S1 = S1+A(I,J)
 40 NEXT J
 60 PRINT "SUM OF ROW" I "=" S1
 65 NEXT I
 70 FOR J = 1 TO N
 71 LET S1=0
 75 FOR I = I TO N
 80 LET S1=S1+A(I,J)
 85 NEXT I
 90 PRINT "SUM OF COLUMN" J "=" S1
 95 NEXT J
200 DATA 3,1,2,3,4,5,6,7,8,9
300 END

READY

RUN
SUM OF ROW 1 = 6
SUM OF ROW 2 = 15
SUM OF ROW 3 = 24
SUM OF COLUMN 1 = 12
SUM OF COLUMN 2 = 15
SUM OF COLUMN 3 = 18
```

6. Predict the output of the following program. Verify your prediction by running the program.

```
 10 DEF FNA(A,B,C,K) = (-B+K*SQR(B↑2-4*A*C))/2*A
 20 READ A,B,C,K
 30 LET Y=FNA(A,B,C,K)
 40 PRINT Y
 50 GOTO 20
200 DATA 1,6,9 , 1
201 DATA 1,-8,16,1
300 END
```

7. Run the programs you wrote in exercises 1-4 above and verify that they are correct.

CHAPTER VII

FORTRAN FOR USERS OF FOCAL

Introduction

FORTRAN is the oldest of algebraic computing languages and it is now used internationally. Despite its faults which are due primarily to the fact that it was a pioneer language, FORTRAN is bound to be with us for a long time to come. Hence it is important that any user of FOCAL be conversant with FORTRAN, if only because FORTRAN has a rich literature and is therefore an important source of algorithms for all computing languages.

This chapter aims at giving the FOCAL user a reading knowledge of FORTRAN as well as an elementary subset of that language in which he may write FORTRAN programs.

FORTRAN as a Batch Processed Language

In most computing systems FORTRAN is used as a batch processed language. That means that FORTRAN programs are written on decks of cards and processed by computing centers in batches. Actually, many computing centers now process individual jobs, entering them into the computer as received where they are first "spooled" or entered into a que on magnetic discs. The computer then reads the jobs (often in accordance with the priority assigned them) from the discs, processes the jobs, and writes their output on other magnietic discs, which are eventually printed on line printers, very rapid printing devices which print one line at a time. The user then obtains his original deck of cards together with his printed output which usually contains a listing of his program, as well as the output the program produced and together with any error messages that the FORTRAN compiler found.

The WATFOR and WATFIV Compilers

Practice has shown that FORTRAN programs may be usefully divided into two classes, which we call here "student jobs" and "production jobs". "Student jobs" designates all programs which, whether written by students or not, are still in the development phase. If they are actually used, they will only be run once or twice, probably not taking much time in the process. "Production jobs", on the other hand, are FORTRAN programs which are already in their final state and which may be used many times over, or which will run for a long time (say in excess of 10 minutes). Since student jobs are short and still undergoing change, it is not important that the "code" or machine language program produced by the compiler be the most efficient possible. What is important is that the error diagnostics be clear and easy to follow, and that the compilation, i.e., the production of the machine code be done very rapidly. Special "Student" compilers have been written which compile FORTRAN very rapidly and with excellent error codes. These compilers were written by members of the computing science department at the University of Waterloo in Canada. They named their first compiler

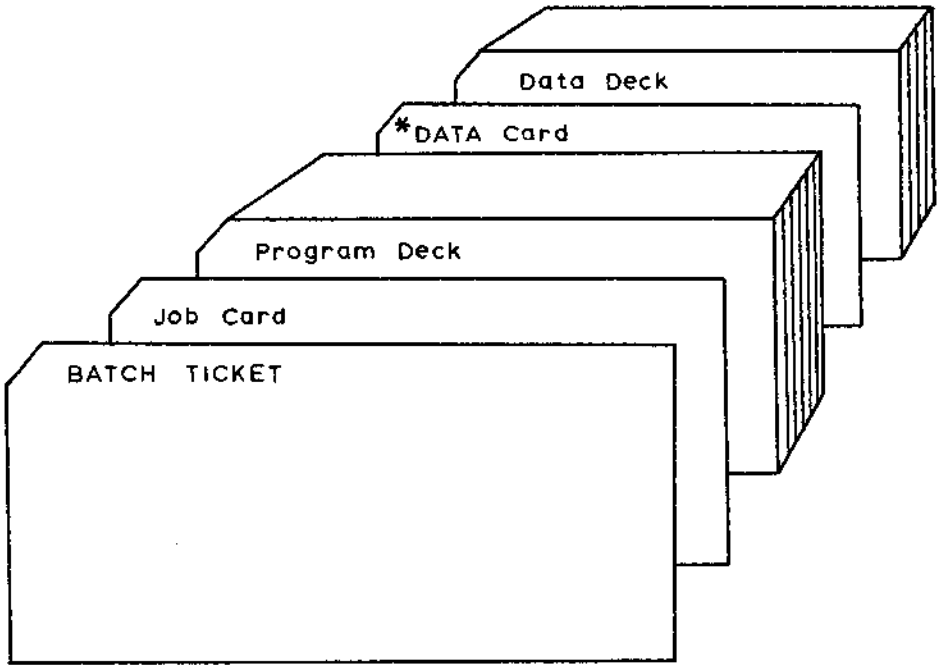


Figure 2. Structure of typical WATFIV deck for batch processed FORTRAN.

WATFOR and the later version WATFIV. WATFOR and WATFIV are currently used to compile student FORTRAN jobs in many American universities. There are more than 300 WATFOR and WATFIV error codes consisting of a prefix and a number. The prefix gives the general type of the error and the following number indicates the specific sort within that genus. A list of WATFOR/WATFIV prefixes is given in appendix 6.

A Subset of WATFOR compiled FORTRAN for FOCAL Users

Input

Since there is no entering variable values from a teletype in a batch processed language, some provision must be made for data to be accessed by the program.

This is accomplished by placing the data at the end of the program deck following a *DATA card. A typical WATFOR deck is shown in Figure 2.

The input command is

```
READ, list
```

where list designates a variable list. When the READ statement is encountered in the program, data will be read from data cards until the list is satisfied. Data may be placed anywhere on a card and as many cards in the data deck as necessary will be read until values for all the variables in the list are found. Every time the program encounters a READ statement data is read beginning from the next card in the data deck.

Output

Variables may be printed out with the statement

```
PRINT, list
```

As in the FOCAL TYPE statement, computation may be performed in the WATFOR PRINT statement and the list may contain algebraic expressions.

Assignment Statement

FORTRAN uses no verb like LET or SET but rather indicates assignment simply with the variable name and the equals sign, e.g. X=Y/2.3.

Statement Numbers

FORTRAN statements may have statement numbers, but they are not needed as the normal order of the program is determined by the card sequence from the front to the back of the deck. Note that there are no compound statements in standard FORTRAN IV.

Types of Variables and Variable Names

There are two important types of numerical variables in FORTRAN, which hinge

on a distinction which neither BASIC or FOCAL makes. These types are INTEGER and REAL variables. REAL variables correspond most closely to the variables in FOCAL or BASIC while an INTEGER variable is a new entity. INTEGER variables may only have (positive or negative) integer values. INTEGER variable names always begin with the letters I through N, and all other letters used as the first letter of a variable name designate REAL variables.

Variable names in FORTRAN IV must begin with a letter (any letter) and may contain up to 7 letters or numbers.

Program Logic

FORTRAN has the classical IF statement

```
IF (expression) j,k,l
```

where "expression" stands for an algebraic expression and j,k, and 1 are statement numbers to which control is transferred if the value of the expression is less than, equal to or greater than 0.

FORTRAN also has a "logical" IF statement of the form

```
IF (logical expression)    command
```

Here a logical expression refers to expressions like X.EQ.Y or Z.LE.X meaning X equals Y or Z is less than or equal to X. If the logical expression is true, then the statement in the IF command following it will be executed. If the logical expression is not true, then the command will be ignored and control will pass to the next statement. Logical relational operators are .LT., .LE., .EQ., .GE., .GT., .NE., standing for, "less than", "less than or equal to", "equal to", "greater than or equal to", "greater than", "not equal to".

The unconditional transfer of control statement GO TO in FORTRAN is the same as FOCAL's GOTO. Note that there is a space between the words in FORTRAN.

Iteration

The FORTRAN iteration statement is the DO command. It has several restrictions which are not common to FOCAL or BASIC. Like BASIC the FORTRAN command uses a terminating statement to indicate the end of the loop. Its form is

```
          DO n index = j,k
                :
                :
             statements within the loop
                :
                :
n         CONTINUE
```

where n is a statement number designating the end of the loop, index is the incremented INTEGER variable and j and k are the positive INTEGER bounds and where j is less than k.

Entry and exit from DO loops is a bit complicated. Loops may be nested, and while one may exit from a loop at any time carrying with him the last value of the index, one may only enter a loop by starting at the beginning with the DO statement.

Subscripts

All subscripted variables must be dimensioned with a DIMENSION statement giving the variable and, in parentheses following it, the maximum value of the subscript or subscripts. E.g.,

```
DIMENSION X(25), ISPY(3,4)
```

Implicit Functions

FORTRAN has a very full complement of internally defined functions (see Appendix 7). These are used just like functions in FOCAL and BASIC, only one must be careful to make sure the argument of the function is the right type (INTEGER or REAL).

Subroutines

Subroutines in FORTRAN have a much more independent existance than those of FOCAL or BASIC. They are really separate programs which are called by the main program, receive values of variables from it, do some computing and pass specific values back to the main program. Unless otherwise specified the only variable names common both to the main program and the subroutine are those listed in the calling statement; moreover, even these need not be the same names in both main program and subroutine! All this separation is to allow the independent writing of subroutines without any regard to the main program or vice versa, thus allowing any subroutine to be used with any FORTRAN program.

As in BASIC one enters a subroutine with a calling statement. In FORTRAN, the form is

```
CALL subroutine name (variable list)
```

where subroutine name is just that and follows the same rules as variable names, and variable list is a list of the variable names as known by the main program which it will share with the subroutine. Note well: these may or may not be the same names for the variables that the subroutine uses. How these names correspond to those used by the subroutine is established by the first statement in the subroutine which has the form

```
Subroutine name (variable list)
```

Here the subroutine name is the same one used by the main program but the variable list, while it corresponds in number and type to the list of the calling statement, uses the subroutine's names for the variables given to it by the main routine. The following example of a subroutine call from a FORTRAN program may illustrate this point.

```
CALL SAMPSON(X,Y,Z)                    (main program)
SUM=Z+3.723


SUBROUTINE SAMPSON (A,B,C)             (entire subroutine)
C=(A/B)*3.14159
RETURN
END
```

When the subroutine is called the values of the subroutines variables A and B will be set equal to X and Y respectively, C will be computed and the main routine will set Z equal to C and go about its business. If ever the variable names A and B appear in the main routine, they have no relation to the variables A and B of the subroutine.

Function Subroutines

A special sort of subroutine forms a user defined function as follows. One appends a function subroutine on the end of his main program like:

```
FUNCTION SALLY (A,B,C)
SALLY=A+B/C
RETURN
END
```

The SALLY function may be used just like a variable name in computation in the main program. Thus, in the main program one might have

```
X=A*SQRT(SALLY(E,F,G))
```

If a function subroutine is short enough to fit on one line it may be abbreviated to a single statement which, like in BASIC, may be included without any other fol de rol as a single statement before it is used in the main program. E.g.,

```
SALLY (A,B,C) = A+B/C
X=A*SQRT(SALLY(E,F,G))
```

Additional FORTRAN Statements

While a knowledge of the statements discussed above will enable the user to write useful FORTRAN programs, there are a number of other statements a user should know about if he is to make sense of the FORTRAN programs he elects to read and translate into an interactive language. These statements are discussed in this section.

FORMAT and Formatted Input and Output Statements

Usually FORTRAN input and output is handled with READ and WRITE statements which also specify the FORMAT of the input or output data. The number of the FORMAT statement controlling the data input or output is given in parentheses

after the READ or WRITE statement along with the device code specifying where in the computing system the data is to be found or printed. For example:

```
READ (5,35) X,Y,Z
```

is an instruction to read data from device 5 according to format statement 30. Somewhere in the program (it could be almost anywhere) one will find a format statement like:

```
3Ø FORMAT (3F1Ø.4)
```

which tells the computer that there are three real variables in the first 30 columns of a card.

FORMAT statements are termed "non-executable" statements. They may be placed almost anywhere in the deck, and referenced by more than one READ or WRITE statement.

COMMON Statements and REAL and INTEGER Declarations

There are a number of statements in FORTRAN which are termed "declarations". A dimension statement is an example. Others are the COMMON statement which is used to over-ride the separation of variables in MAIN and subroutines and the REAL and INTEGER statements which may over-ride the implicit variable TYPE convention given above. These latter statements may also be used to dimension variables and to set their initial values.

Another statement used to define initial values of variables in a program is the DATA statement which contains a list of variables followed by the values to which the variables should be set.

With these guidelines you should be able to write simple FORTRAN programs and translate even more complicated programs into FOCAL or BASIC. The following exercises will afford you an opportunity for practice. You may wish to consult one of the standard FORTRAN manuals listed in the bibliography for help.

Exercise Set 8

Paper and Pencil

1. The following FORTRAN program and its subroutine reverses a portion of a vector, B, between the limits L and LL. (From "A FORTRAN IV Primer," Organick, p. 155.) Translate the entire program into FOCAL.

```
C       A PROGRAM TO TEST THE REVERSE SUBROUTINE
        DIMENSION B (200)
   20   FORMAT (3I5/(6F10.5))
   30   FORMAT (6F12.5)
   50   READ (5,20) L,LL,MN,  (B(I) , I = 1,MN)
        CALL REVERSE (B,L,LL)
        WRITE (6,30)  (B(I), I=1,MN)
        GO TO 50
        END

        SUBROUTINE REVERSE (A,M,N)
        DIMENSION A (200)
        MNP = M + N
        MIDDLE = MNP/2
        DO 10 I = M,MIDDLE
        K = MPN-I
C       REVERSE THE PAIR CONSISTING OF
C       A(I) AND A(M+N-I)
        T=A(I)
        A(I) = A(K)
        A(K) = T
   10   CONTINUE
        RETURN
        END
```

Note that DIMENSION statements need not be translated. Statement 50 and the WRITE statement below it contain an implied DO loop, namely: (B(I), I=1,MN) a form allowed in input and output statements. The effect of the statement is to cause the reading or writing of all the values of B(I) from 1 to MN at one time.

2. The following is a routine to evaluate the definite integral of a function, FUN, from A to B (from "Introduction to FORTRAN IV Programming Using the WATFOR Compiler", Blatt, page 169). Translate it into FOCAL, and supply it with a function to integrate. Check the routine against a function whose integral between limits you know.

```
      FUNCTION SIMPSN(FUN,A,B,N)
C     SIMPSON'S RULE INTEGRAL
C     FUN = FUNCTION NAME
```

```
C       A = INITIAL VALUE OF X
C       B = FINAL VALUE OF X
C       N = NUMBER OF INTERVALS, EVEN
C       EVALUATE INTEGRAL OF FUN(X) FROM
C       X = A TO X = B, USING N INTERVALS"
C
C       ENSURE EVEN NUMBER OF INTERVALS .GE. 2
C
 1000   M=(N/2)*2
        IF (M.GE.2)  GO TO 1050
C       CRAZY ENTRY.  RETURN ZERO VALUE
        SIMPSN=0.0
        RETURN
C       ENTRY O.K.  SET UP STUFF

 1050   H=(B-A)/FLOAT(M)
        EVNSUM=0.0
        ODDSUM=0.0
        ENDSUM=FUN (A)+FUN(B)

        X=A+H
        MMIN1=M-1
        J=1
C       THIS J IS A SWITCH.  J=1 MEANS AN ODD POINT, J=2 AN EVEN PT.
C       NOW START THE LOOP
        DO 1200 I=1, MMINI
        IF (J-1) 1300, 1101,1102
C       I IS AN ODD POINT
 1101   ODDSUM=ODDSUM +FUN(X)
        J=2
        GO TO 1150
C       I IS AN EVEN POINT
 1102   EVNSUM =EVNSUM +FUN(X)
        J=2
C       PATHS REJOIN
 1150   X=X+H
 1200   CONTINUE
C       ALL OVER BUT THE SHOUTING
        SIMPSN=(H/3.)*(ENDSUM+4.0*ODDSUM+2.0*EVNSUM)
 1300   RETURN
        END
```

3. Look up Simpson's rule for evaluating definite integrals in Conte's "Elementary Numerical Analysis", or a similar book and verify that the program given in Blatt is correct.

Computer

4. Run and de-bug the programs of exercises 1 and 2 above.

CHAPTER VIII

FINDING AND APPLYING COMPUTING ALGORITHMS

Introduction

It was pointed out above that computing is an extremely fast-growing and fast-changing field. The introduction of algebraic computing languages is having a great impact on the teaching of mathematics, a discipline which has changed very slowly in the past. We are, however, still in a transition period and the literature of mathematics has yet to catch up. This means that the user must often spend some time searching for the algorithms he needs to solve his problems.

Numerical Analysis

The field of study in colleges and universities which deals with the heart of the mathematics of digital computing is called "Numerical Analysis". The major topics of a course in Numerical Analysis might be:

Solution of implicit equations (like $\sin x = x+x^{1/3}$)
Curve fitting
Interpolation
Numerical solution of differential equations
Numerical integration
Matrix manipulations

Most modern books of numerical analysis are oriented to the use of digital computers and some include actual FORTRAN programs as examples. E.g. see Conte's book, Elementary Numerical Analysis, An Algorithmic Approach, McGraw Hill, 1965.

In general it may be said that numerical methods employ finite approximations to problems involving continuous variables. The difficulty of the subject lies in the justification of approximations used and the evaluation of the errors that they introduce.

If the user finds he has a problem touching any one of the above topic areas listed he would do well to consult a standard text in numerical analysis before proceeding further.

This point should be emphasized: there is absolutely no doubt that many persons, with a good knowledge of FOCAL or BASIC, could, all by themselves, write programs for curve fitting, definite integration, etc. But it would be absolutely foolish to do so without consulting a text on the numerical analysis first. To launch out on your own without consulting the literature is often termed "re-inventing the wheel" in computing circles.

Statistics

Many statistics courses are still being taught with the idea that the work

will be performed on desk calculators. This is, of course, exactly the drudgery that digital computers are designed to eliminate and so computing routines for standard statistical tests are common. Many statistical tests are so simple that it is easier for an experienced programmer to write his own version of the test directly from the algorithms given in a text book than to try to wade through someone else's ill-documented program, and hence the entire human race seems destined to continue to write statistical routines for all time to come. However, when a programmer finds he is writing his own routines over and over again, it is time to do something about his documentation habits.

Because statistics books have long been written with the idea that someone will in fact use the tests described in them, they are usually easy to read and provide a valuable source of statistical algorithms.

IBM Scientific Subroutines

The International Business Machines Corporation publishes a set of FORTRAN routines entitled simply "IBM Scientific Subroutines". These are very professionally written FORTRAN subroutines designed to handle a wide variety of computing tasks, including statistical tests. If one is writing a FORTRAN program of some generality, this is the palce to start, and these routines may also afford useful material for translation. While their documentation leaves something to be desired, they are nonetheless some of the best programs available.

Computing Manuals

A growing number of elementary computing manuals are oriented towards particular fields and afford a variety of program examples. Particularly good in this respect is Kemeny and Kurtz's book on BASIC which gives a broad spectrum of programs in elementary mathematics.

Company Users Societies

Many computing companies sponsor users' organizations which collect, publish, and share computing routines. Such a society is DECUS, the users' organization of the Digital Equipment Corporation, which collects and publishes FOCAL programs in their magazine, DECUSCOPE.

APPENDIX 1

Functions Defined in FOCAL*

FOCAL FUNCTION	Mathematical Equivalent
FSQT(A)	$+\sqrt{A}$ defined only for $A \geqq 0$
FABS(A)	$\lvert A \rvert$
FSGN(A)	+1, 0, or -1 depending on the value of mathematical sign of A
FITR(A)	integer part of A, (no roundoff)
FEXP(A)	e^A
FSIN(A)	sin A, A given in radians
FCOS(A)	cos A, A given in radians
FATN(A)	arc tan A, value given in radians
FLOG(A)	$\log_e A$

Useful Trigonometric Functions

FSIN(A)/FCOS(A)	tan A
FATN(A/FSQT(1-A↑2)	arc sin A
FATN(FSQT(1-A↑2)/A)	arc cos A

*Analog input and output functions omitted.

APPENDIX 2

FOCAL Commands, Symbols and Abbreviations

Command	Abbreviation
TYPE	T
ASK	A
WRITE	W
SET	S
IF	I
MODIFY	M
QUIT	Q
DO	D
RETURN	R
COMMENT, CONTINUE	C
ERASE	E
FOR	F
GO, GOTO	G

Other Symbols:

Carriage return and line feed	!
Carriage return only	#
Rubout	"RUBOUT"
Erase all preceeding statement line	←
Statement delimiter for compound statements	;

APPENDIX 3

FOCAL Error Diagnostics for PDP-8, FOCAL-8 1969

Error Code	Meaning
?00.00	Manual start from console.
?01.00	Interrupt from keyboard via Control C.
?01.40	Illegal step or line number.
?01.78	Group number is too large.
?01.96	Double periods found in a line number.
?01.:5	Line number is too large.
?01.;4	Group zero is an illegal line number.
?02.32	Non-existant group referenced by "DO".
?02.52	Non-existant line referenced by "DO".
?02.79	Storage filled by push-down list. (Bad news; your program is probably too long or you are accidentally generating an infinite array.)
?03.05	Non-existant line used after "GOTO" or "IF".
?03.28	An illegal command was used.
?04.39	Part of expression to left of " in error in FOR or SET command.
?04.52	Excess right terminators encountered.
?04.60	Illegal terminator in FOR command.
?04.:3	Missing argument in display command.
?05.48	Incorrect argument in MODIFY command.
?06.06	Illegal use of function or number.
?06.54	Storage is filled by variables. (See comment under 02.79.)

Appendix 3 continued:

?07.22	Operator missing in expression or double E.
?07.38	No operator encountered before parenthesis.
?07.:9	No argument given after function call.
?07.;6	Illegal function name or double operators.
?08.47	Parentheses do not match.
?09.11	Bad argument in ERASE.
?10.:5	Storage is filled with text (see note under 02.79).
?11.35	Input buffer overflow. (This happens if there are not enough spaces in input text, or if input patch to correct this condition is not implemented.)
?20.34	Logarithm of zero is requested.
?23.36	Literal number is too large.
?26.99	Exponent is too large or negative.
?28.73	Division by zero requested.
?30.05	Imaginary square root required.
?31. 7	Illegal character, unavailable command or unavailable function used.

Note: The format for reporting errors in an indirect program is, e.g., ?08.47 @ 01.32 meaning "Parentheses do not match in line 1.32." Often errors in a program subsection called by a DO will be referred to the line number of the statement which called the subsection.

APPENDIX 4

Error Diagnostics for the PDP-15, FOCAL

Error Code	Meaning
?00	Function not implemented.
?01	Illegal character at beginning of line.
?02	Group number illegal as line number.
?03	Group number too large.
?04	Illegal type/ ask format.
?05	Too many periods.
?06	Line number too large.
?07	Line number missing.
?08	Illegal group number.
?09	Push down list overflow.
?10	Illegal command.
?11	Illegal "IF" format.
?12	Left of equals sign in error on FOR or SET.
?13	Excess right parentheses.
?14	Illegal FOR format.
?15	Illegal variable name.
?16	Text/ variable buffer overflow.
?17	Illegal expression format.
?18	Operator missing before parenthesis.
?19	Missing left parenthesis.
?20	Illegal function name.

Appendix 4 continued:

?21	Double operator.
?22	Parenthesis error.
?23	ERASE or "WRITE" argument error.
?24	Negative line number.
?25	Zero argument for log.
?26	Input overflow.
?27	Number too large.
?28	Negative power illegal.
?29	Division by zero illegal.
?30	Square root of a negative number.
?31	Illegal command during library output.
?32	Illegal library command.
?33	Illegal file name.
?34	File not found.
?35	No library output file open.
?36	.OTS error from FORTRAN IV arithmetic package.
?37	COMMON format error.

APPENDIX 5

BASIC Error Messages for PDP-8 Edusystem 20

WHAT?	Command not understood - ready mode.
ERROR 1	Log of negative or zero number requested.
ERROR 2	Square root of negative number requested.
ERROR 3	Division by zero requested.
ERROR 4	Overflow - exponent greater than approximately +38.
ERROR 5	Underflow - exponent less than approximately -38.
ERROR 6	Line too long or program too big.
ERROR 7	Characters are being typed in too fast - use TAPE command for reading paper tapes.
ERROR 8	System overload caused character to be lost.
ERROR 9	Program too complex or too many variables.
ERROR 10	Missing or illegal operand or double operators.
ERROR 11	Missing operator before a left parentheses.
ERROR 12	Missing or illegal number.
ERROR 13	Too many digits in number.
ERROR 14	No DEF for function call.
ERROR 15	Missing or mismatched parentheses or illegal dummy variable in DEF.
ERROR 16	Wrong number of arguments in DEF call.
ERROR 17	Illegal character in DEF expression.
ERROR 18	Missing or illegal variable.
ERROR 19	Single and double subscripted variables with the same name.

Appendix 5 continued:

ERROR 20	Subscript out of range.
ERROR 21	No left parenthesis in function.
ERROR 22	Illegal user defined function - not FN followed by a letter and a left parenthesis.
ERROR 23	Mismatched parentheses or missing operator after right parenthesis.
ERROR 24	Syntax in GOTO.
ERROR 25	Syntax in RESTORE.
ERROR 26	Syntax in GOSUB.
ERROR 27	Syntax in ON.
ERROR 28	Index out of range in ON.
ERROR 29	Syntax in RETURN.
ERROR 30	RETURN without GOSUB.
ERROR 31	Missing left parenthesis in TAB function.
ERROR 32	Syntax in PRINT.
ERROR 33	No END statement or END is not the last statement.
ERROR 34	Missing or illegal line number.
ERROR 35	Attempt to GOTO or GOSUB to a non-existent line.
ERROR 36	Missing or illegal relation in IF.
ERROR 37	Syntax in IF.
ERROR 38	Missing equal sign or improper variable left of the equal sign in LET or FOR.
ERROR 39	Subscripted index in FOR.
ERROR 40	Syntax in FOR.
ERROR 41	No NEXT for FOR.
ERROR 42	Syntax in LET.

ERROR 43	Syntax in NEXT.
ERROR 44	NEXT without FOR.
ERROR 45	Too much data typed in or illegal character in DATA or the data typed in.
ERROR 46	Illegal character or function in INPUT or READ.
ERROR 47	Out of data.
ERROR 48	Unrecognized command - RUN mode.

APPENDIX 6

WATFOR and WATFIV Error Prefixes

Prefix	Error Location
AL	Assembler language subprogram
AS	Assign statement
BD	Block Data Statement
CC	Card format and contents
CM	Common statement
CN	FORTRAN Constants
CP	Compiler error
CV	Character Variable
DA	Data Statement
DF	Define File Statement
DM	Dimension Statement
DO	Do loop
EC	Equivalence and or Common
EN	End Statement
EQ	Equal signs
EV	Equivalence Statements
EX	Powers and Exponentiation
EY	Entry Statement
FM	Format
FN	Functions and Subroutines
FT	Format

Appendix 6 continued:

GO	GO TO Statements
HO	Hollerith Constants
IF	IF Statement (Arithmetic and Logical)
IM	IMPLICIT Statement
IO	Input/Output
JB	Job Control cards
KO	Job termination
LG	Logical operations
LI	Library routines
MD	Mixed Mode
MO	Memory Overflow
NL	Namelist statements
PC	Parentheses
PS	Pause & Stop Statements
RE	Return statement
SF	Arithmetic and Logical statement functions
SR	Subroutines
SS	Subscript
ST	Statements and statement number
SV	Subscripted variable
SX	Syntax Error
TY	Type statement
UN	I/O Operations
UV	Undefined variable

APPENDIX 6

VA	Variable Names
XT	External Statement

*A WATFOR or WATFIV error code contains a prefix and a number, e.g., KO-6, which means "Job time limit exceeded." For a complete list of error codes, consult a WATFOR manual or your computing center.

APPENDIX 7

Commonly Used Implicit Functions (FORTRAN)

Function	Definition	Name*	Type+ of Argument	Function
Absolute value	arg	ABS	R	R
Exponential	e^{arg}	EXP	R	R
Natural log	ln (arg)	ALOG	R	R
Common log	$\log_{10}$ (arg)	ALOG10	R	R
Sine	sin (arg)	SIN	R	R
Cosine	cos (arg)	COS	R	R
Tangent	tan (arg)	TAN	R	R
Arctangent	$\tan^{-1}$ (arg)	ATAN	R	R
Acctangent	$\tan^{-1}$ (arg/arg)	ATAN2	R,R	R
Square Root	$arg^{\frac{1}{2}}$	SQRT	R	R
Truncation	Sign of argument times largest integer of arg	AINT	R	R
		INT	R	I
Transfer of sign	Sign of arg_2 times absolute value of arg_1	SIGN	R	R
		ISIGN	I	I
FIX	Conversion from REAL to INTEGER	IFIX	R	I
Float	Conversion from INTEGER to REAL		I	R
Remaindering	arg_1 mod arg_2	AMOD	R	R
		MOD	I	I
Max	Choosing largest value	AMAX1	R,R,...	R
		MAXO	I,I,I...	I
MIN	Choosing the minimum value	AMIN1	R,R,...	R
		MINO	I,I,I...	I

* The argument of the function always appears in parentheses after the name.

+ R = REAL, I = INTEGER

BIBLIOGRAPHY OF COMPUTING BOOKS

Books on FOCAL

Programming Languages, Volume 2. PDP-8 Handbook Series, Digital Computer Corporation. Maynard, Massachusetts, 1970.

Books about BASIC

Kemeny, J.G., Kurtz, T.E. BASIC Programming. John Wiley and Sons Inc. New York, 1968.

Sharpe, W.F., Jacob, et al. BASIC, An Introduction to Computer Programming Using the BASIC Language, Revised Edition. The Free Press. New York, 1971.

Spencer, D.D. A Guide to BASIC Programming: A Time-Sharing Language. Addison-Wesley Publishing Company. Reading, Massachusetts. 1970.

Books about FORTRAN

Kennedy, M., Solomon, M.B. Ten Statement FORTRAN Plus FORTRAN IV. Prentice-Hall Inc. Englewood Cliffs, New Jersey, 1970.

Kress, P., Dirksen, P., Graham, J.W. FORTRAN IV with WATFOR.Englewood Cliffs, New Jersey, 1968.

Books about Programming Languages in General

Higman, B. A Comparative Study of Programming Languages. MacDonald: London and American Elsevier Inc. New York, 1967.

Jordain, P.B., Ed. Condensed Computer Encyclopedia. McGraw Hill Book Company. New York, 1969.

Rosen, S., Ed. Programming Systems and Languages. McGraw Hill Book Company. New York, 1967.

Books about Numerical Analysis

Conte, S.D. Elementary Numerical Analysis, An Algorithmic Approach. McGraw-Hill Book Company. New York, 1965.

Bibliography, continued:

Hamming, R.W. Numerical Methods for Scientists and Engineers. McGraw Hill Book Company. New York, 1962.

Books Containing Useful Algorithms

Abramowitz, M. and Stegun, I.A., eds. Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables. U.S. Department of Commerce, National Bureau of Standards, Washington, D.C. 9th printing, 1970.

Beyer, W.H., ed. Handbook of Tables for Probability and Statistics. The Chemical Rubber Co. Cleveland, Ohio, 1966.

Jahnke, Eugene, Emde, Fritz. Tables of Functions, 4th Edition. Dover Publications. New York, 1945.

Comprehensive FORTRAN IV Manual

Organick, E.I. A FORTRAN IV Primer. Addison-Wesley Publishing Co. Reading, Massachusetts, 1968.

INDEX

Principal reference is given first.