CURSO: SEMINARIO DE FORTRAN 77
CURSO DE SUPERACION ACADEMICA.

EN COLABORACION CON LA DIVISION DE CIENCIAS BASICAS, DENTRO DEL PROGRAMA DE SUPERACION DEL PERSONAL ACADEMICO DE LA FACULTAD DE INGENIERIA.

22 AL 25 DE MARZO DE 1983

APPENDIX C

FORTRAN LANGUAGE SUMMARY

C.1 EXPRESSION OPERATORS

The following lists the expression operators in each data type in order of descending procedence:

Data Type	Operator	Operation	Operates upon:
Arithmetic	**	Exponentiation	Arithmetic or logical expressions
	*,/	Multiplication, division	
	+,-	Addition, subtraction, unary plus and minus	
Character	//	Concatenation	Character expressions
Relational	.GT.	Greater than	Arithmetic, logical, or character expressions (all relational operators have equal precedence)
	.GE.	Greater than or equal to	
	.LT.	Less than	
	.LE.	Less than or equal to	
	.EQ.	Equal to	
	.NE.	Not equal to	
Logical	.NOT.	.NOT.A is true if and only if A is false	Logical or integer expressions
	.AND.	A.AND.B is true if and only if A and B are both true	

Data Type	Operator	Operation	Operates upon:
Logical (Cont.)	.OR.	A.OR.B is true if either A or B or both are true	
	.EQV.	A.EQV.B is true if and only if A and B are both true or A and B are both false	.EQV., .NEQV., and .XOR. have equal priority
	.XOR.	A.XOR.B is true if and only if A is true and B is false or B is true and A is false	
	.NEQV.	Same as .XOR.,	

C.2 STATEMENTS

The following summarizes the statements available in the VAX-11 FORTRAN language, including the general form of each statement. The statements are listed alphabetically for ease of reference. The "Manual Section" column indicates the section of this manual that describes each statement in detail.

Form	Effect	Manual Section
ACCEPT	See READ.	7.7
Arithmetic/Logical/Character Assignment		3.1, 3.2, 3.3

v=e

v — is a variable name, an array element name, or a character substring name.

e — is an expression.

Assigns the value of the arithmetic, logical, or character expression to the variable.

ASSIGN s TO v — 3.4

s — is the label of a FORMAT statement or an executable statement.

v — is an integer variable name.

Associates the statement label s with the integer variable v for later use as a format specifier or in an assigned GO TO statement.

Form	Effect	Manual Section

BACKSPACE ([UNIT=]u[,IOSTAT=ios][,ERR=s]) 9.5
BACKSPACE u

u — is a logical unit specifier.

ios — is an integer variable or integer array element.

s — is the label of an executable statement

Backspaces the currently open file on logical unit u one record.

BLOCK DATA [nam] 5.12

nam — is a symbolic name.

Specifies the subprogram that follows as a BLOCK DATA subprogram.

CALL f[([a][,[a]]...)] 4.6, 6.2

f — is a subprogram name or entry point.

a — is an expression, an array name, a procedure name, or an alternate return specifier. An alternate return specifier is *s or &s, where s is the label of an executable statement.

Calls the subroutine subprogram with the name specified by f, passing the actual arguments a to replace the dummy arguments in the subroutine definition.

CLOSE ([UNIT=]u[,p][,IOSTAT=ios][,ERR=s]) 9.2

p — is one the following parameters:

```
STATUS     'SAVE'
DISPOSE =  'KEEP'
DISP       'DELETE'
           'PRINT'
           'SUBMIT'
           'PRINT/DELETE'
           'SUBMIT/DELETE'
```

u — is a logical unit specifier.

s — is the label of an executable statement.

ios — is an integer variable or integer array element.

Closes the specified file.

Form	Effect	Manual Section
COMMON [/[cb]/] nlist [[,]/[cb]/nlist]...		5.4
	cb — is a common block name.	
	nlist — is a list of one or more variable names, array names, or array declarators separated by commas.	
	Reserves one or more blocks of storage space under the name specified to contain the variables associated with that block name.	
CONTINUE		4.5
	Causes no processing.	
DATA nlist/clist/[[,] nlist/clist/]...		5.9
	nlist — is a list of one or more variable names, array names, array element names, character substring names, or implied DO lists, separated by commas. Subscript expressions and substring expressions must be constant.	
	clist — is a list of one or more constants separated by commas, each optionally preceded by j*, where j is a nonzero, unsigned integer constant.	
	Initially stores elements of clist in the corresponding elements of nlist.	
DECODE (c,f,b[,IOSTAT=ios][,ERR=s])[list]		A.1
	c — is an integer expression.	
	f — is a format specifier.	
	b — is a variable name, array name, array element name, or character substring name.	
	ios — is an input/output status specifier.	
	s — is a label of an executable statement.	
	list — is an I/O list.	
	Reads c characters from buffer b and assigns values to the elements in the list converted according to format specification f.	

Form	Effect	Manual Section

DEFINE FILE u(m,n,U,v)[,u(m,n,U,v)]...

A.2

u — is a logical unit specifier.

m — is a constant or variable.

n — is a constant or variable.

U — specifies unformatted.

v — is an integer variable name.

Defines the record structure of a direct access file where u is the logical unit number, m is the number of fixed-length records in the file, n is the length in 16-bit words of a single record, U is a fixed argument, and v is the associated variable.

DELETE ([UNIT=]u[,REC=r][,IOSTAT=ios][,ERR=s])
DELETE (u'r[,IOSTAT=ios][,ERR=s])

9.7

u — is a logical unit specifier.

r — is a relative record number specifier.

ios — is an input/output specifier.

s — is the label of an executable statement.

Deletes records from relative or indexed files, where u is the logical unit connected to the file, r is the number of the record in a relative file, ios is an input/output status specifier, and s is the label of the statement to which control is to be transferred if an error occurs.

DIMENSION a(d)[,a(d)]...

5.3

a(d) — An array declarator.

Specifies storage space requirements for arrays.

Form	Effect	Manual Section

DO [s [,]] v = e1,e2[,e3] — 4.3.1

s — is the label of an executable statement.

v — is a variable name.

e1,e2,e3 are numeric expressions.

Executes the DO loop by performing the following steps:

1. Evaluates cnt= INT((e2-e1+e3)/e3)
2. Sets v=e1
3. If cnt is less than or equal to zero, does not execute the loop
4. If cnt is greater than zero, then
   a. Executes the statements in the body of the loop
   b. Evaluates v=v+e3
   c. Decrements the loop count (cnt=cnt-1). If cnt is greater than zero, repeats the loop

DO [s[,]]WHILE(e) — 4.3.2

s — is a statement label.

e — is a logical expression.

Similar to the DO statement, but executes as long as the logical expression contained in the statement continues to be true, instead of for a specified number of iterations.

ELSE — 4.2.3

Defines a block of statements to be executed if logical expressions in previous IF THEN and ELSE IF THEN statements have values of false. See IF THEN.

Form	Effect	Manual Section
ELSE IF (e) THEN		4.2.3
	e — is a logical expression.	
	Defines a block of statements to be executed if logical expressions in previous IF THEN and ELSE IF THEN statements have values of false, and the logical expression e has a value of true. See IF THEN.	
ENCODE (c,f,b[,IOSTAT=ios][,ERR=s])[list]		A.1
	c — is an integer expression.	
	f — is a format specifier.	
	b — is a variable name, array name, array element name, or substring name.	
	ios — is an input/output specifier.	
	s — is a label of an executable statement.	
	list — is an I/O list.	
	Writes c characters into buffer b, which contains the values of the elements of the list, converted according to format specification f.	
END		4.10
	Delimits a program unit.	
END DO		4.4
	May be used in place of a labeled statement to delimit the body of a DO loop.	
ENDFILE ([UNIT=]u[,IOSTAT=ios][,ERR=s]) ENDFILE u		9.6
	u — is a logical unit specifier.	
	ios — is an input/output specifier.	
	s — is the label of an executable statement.	
	Writes an end-file record on logical unit u.	

Form	Effect	Manual Section

END IF — 4.2.3

Terminates block IF construct. See IF THEN.

ENTRY nam [([p[,p]...])] — 6.2.4

nam — is a subprogram name.

p — Is a symbolic name or an alternate return specifier(*).

Defines an alternate entry point within a subroutine or function subprogram.

EQUIVALENCE (nlist)[,(nlist)]... — 5.5

nlist — is a list of two or more variable names, array names, array element names, or character substring names separated by commas. Subscript expressions and substring expressions must be compile-time constant expressions.

Assigns each of the names in nlist the same storage location.

EXTERNAL v[,v]... — 5.7, A.6

v — is a subprogram name.

Defines the names specified as user-defined subprograms.

FIND ([UNIT=]u,REC=r[,IOSTAT=ios][ERR=s]) — A.3
FIND (u'r[,IOSTAT=ios][,ERR=s])

u — is a logical unit specifier.

r — is an integer expression.

ios — is an input/output specifier.

s — is the label of an executable statement.

Positions the file on logical unit u to record r and sets the associated variable to record number r.

Form	Effect	Manual Section

FORMAT (field specification,...) 8.1 - 8.7

Describes the format in which one or more records are to be transmitted; a statement label must be present.

[typ] FUNCTION nam[*n][([p[,p]...])] 6.2.2

typ is a data type specifier.

nam is a symbolic name.

*n is a data type length specifier.

p is a symbolic name.

Begins a function subprogram, indicating the program name and any dummy argument names (p). An optional type specification can be included.

GO TO s 4.1.1

s is a label of an executable statement.

Transfers control to statement number s.

GO TO (slist)[,] e 4.1.2

slist is a list of one or more statement labels separated by commas.

e is an integer expression.

Transfers control to the statement specified by the value of e (if e=1, control transfers to the first statement label; if e=2, control transfers to the second statement label, and so forth). If e is less than 1 or greater than the number of statement labels present, no transfer takes place.

GO TO v [[,](slist)] 4.1.3

v is an integer variable name.

slist is a list of one or more statement labels separated by commas.

Transfers control to the statement most recently associated with v by an ASSIGN statement.

Form	Effect	Manual Section

IF (e) s1,s2,s3 — 4.2.1

e — is an expression.

s1,s2,s3 are labels of executable statements.

Transfers control to statement si depending on the value of e (if e is less than zero, control transfers to s1; if e equals zero, control transfers to s2; if e is greater than zero, control transfers to s3).

IF (e) st — 4.2.2

e — is an expression.

st — is any executable statement except a DO, END DO, END, block IF, or logical IF.

Executes the statement if the logical expression has a value of true.

IF (e1) THEN — 4.2.3

block

ELSE IF (e2) THEN

block

ELSE

block

END IF

e1,e2 — are logical expressions.

block — is a series of zero or more FORTRAN statements.

Defines blocks of statements and conditionally executes them. If the logical expression in the IF THEN statement has a value of true, the first block is executed and control transfers to the first executable statement after the END IF statement.

Form	Effect	Manual Section

If the logical expression has a value of false, the process is repeated for the next ELSE IF THEN statement. If all logical expressions have values of false, the ELSE block is executed. If there is no ELSE block, control transfers to the next executable statement following END IF.

IMPLICIT typ (a[,a]...)[,typ(a[,a]...)]... 5.1

typ is a data type specifier.

a is either a single letter, or two letters in alphabetical order separated by a hyphen (that is, X-Y).

The element a represents a single (or a range of) letter(s) whose presence as the initial letter of a variable specifies the variable to be of that data type.

INCLUDE 'file specification[/[NO]LIST]' 1.5

'file specification'
is a character constant.

Includes the source statements in the compilation from the file specified.

INQUIRE (par[,par]...) 9.3

par is a keyword specification having the form

key=variable

key is a keyword as described below.

value depends on the keyword.

Keyword	Values
Inputs	
FILE	fin
UNIT	e

Form	Effect	Manual Section

INQUIRE (par[,par]...) (Cont.)

outputs

ACCESS	cv
BLANK	cv
CARRIAGECONTROL	cv
DIRECT	cv
ERR	s
EXIST	lv
FORM	cv
FORMATTED	cv
IOSTAT	v
KEYED	cv
NAMED	lv
NEXTREC	v
NUMBER	v
OPENED	lv
ORGANIZATION	cv
RECL	v
RECORDTYPE	cv
SEQUENTIAL	cv
UNFORMATTED	cv

e — is a numeric expression.

fin — is a character expression.

v — is an integer variable or integer array element.

lv — is a logical variable or array element.

cv — is a character variable, array element, or substring reference.

s — is the label of an executable statement.

Furnishes information on specified characteristics of a file or of a logical unit and its connections to a file.

INTRINSIC fun [,fun]... 5.8

fun — is an intrinsic function name.

Identifies symbolic names as representing intrinsic functions and allows those names to be used as actual arguments.

Form	Effect	Manual Section

OPEN(par[,par]...) 9.1

par is a keyword specification in one of the following forms:

```
key
key = value
```

key is a keyword, as described below.

value depends on the keyword.

Keyword	Values
ACCESS	'SEQUENTIAL' 'DIRECT' 'KEYED' 'APPEND'
ASSOCIATEVARIABLE	v
BLOCKSIZE	e
BLANK	'NULL' 'ZERO'
BUFFERCOUNT	e
CARRIAGECONTROL	'FORTRAN' 'LIST' 'NONE'
DISP	(same as DISPOSE)
DISPOSE	'KEEP' or 'SAVE' 'PRINT' 'DELETE' 'SUBMIT' 'SUBMIT/DELETE' 'PRINT/DELETE'
ERR	s
EXTENDSIZE	e
FILE	c
FORM	'FORMATTED' 'UNFORMATTED'
INITIALSIZE	e
IOSTAT	v
KEY	keyspec
MAXREC	e
NAME	(Same as File)
NOSPANBLOCKS	-
ORGANIZATION	'SEQUENTIAL' 'RELATIVE' 'INDEXED'
READONLY	-
RECL	e
RECORDSIZE	(same as RECL)
RECORDTYPE	'FIXED' 'VARIABLE' 'SEGMENTED'
SHARED	-
STATUS	'OLD' 'NEW' 'SCRATCH' 'UNKNOWN'
TYPE	(same as STATUS)
UNIT	e
USEROPEN	p

Form	Effect	Manual Section
	c is a character expression, numeric array name, numeric variable name, numeric array element name, or Hollerith constant.	
	e is a numeric expression.	
	p is a program unit name.	
	s is a statement label.	
	v is an integer variable name.	
	keyspec is (e1:e2[:type]),	
	where:	
	e1 is the beginning byte of the key field	
	e2 is the ending byte of the key field	
	type is either INTEGER or CHARACTER	
	Opens a file on the specified logical unit according to the parameters specified by the keywords.	
PARAMETER (p=c [,p=c]...)		5.10, A.4
	p is a symbolic name.	
	c is a constant or compile-time constant expression.	
	Defines a symbolic name for a constant.	
PAUSE [disp]		4.8
	disp is a decimal digit string containing 1 to 5 digits or a character constant	
	Suspends program execution and prints the display, if one is specified.	
PRINT	See WRITE	7.8
PROGRAM nam		5.11
	nam A symbolic name.	
	Specifies a name for the main program.	

Form	Effect	Manual Section
READ ([UNIT=]u,[FMT=]f[,IOSTAT=ios][,END=s][,ERR=s])[list]		7.4.1
READ f[,list]		7.4.1
ACCEPT f[,list]		7.8

u — is a logical unit specifier.

f — is a format specifier.

ios — is an input/output specifier.

s — is a label of an executable statement.

list — is an I/O list.

Reads one or more logical records from unit u and assigns values to the elements in the list. The records are converted according to the format specifier (f).

READ ([UNIT=]u,[FMT=]*[,IOSTAT=ios][,END=s][,ERR=s])list		7.4.1.2
READ *,[list]		7.4.1.2
ACCEPT *,[list]		7.7

u — is a logical unit specifier.

* — denotes list-directed formatting.

s — is a label of an executable statement.

list — is an I/O list.

Reads one or more logical records from unit u and assigns values to the elements in the list. The records are converted according to the data type of the list element.

READ([UNIT=]u[,IOSTAT=ios][,END=s][,ERR=s])[list]	7.4.1.3

u — is a logical unit specifier.

ios — is an input/output specifier.

s — is a label of an executable statement.

list — is an I/O list.

Reads one unformatted record from unit u and assigns values to the elements in the list.

Form	Effect	Manual Section
READ ([UNIT]=u,[FMT=]f,REC=r[,iostat=ios][,ERR=s])[list]		7.4.2.1
READ (u'r,[FMT=]f[,IOSTAT=ios][,ERR=s])[list]		7.4.2.1
u	is a logical unit specifier.	
r	is a relative record number specifier.	
f	is a format specifier.	
ios	is an input/output specifier.	
s	is a label of an executable statement.	
list	is an I/O list	
	Reads record r from unit u and assigns values to the elements in the list. The record is converted according to f.	
READ([UNIT=]u,REC=r[,IOSTAT=ios][,ERR=s])[list]		7.4.2.2
READ(u'r[,ERR=s])[list]		7.4.2.2
u	is a logical unit specifier.	
r	is a relative record number specifier.	
ios	is an input/output specifier.	
s	is a label of an executable statement.	
list	is an I/O list.	
	Reads record r from unit u and assigns values to the elements in the list.	
READ ([UNIT=]u, [FMT=]f,keyspec[,KEYID=kn][,IOSTAT=ios] [,ERR=s])[list]		7.4.3
READ ([UNIT=]u, keyspec[,KEYID=kn][,IOSTAT=ios] [,ERR=s])[list]		
u	is a logical unit specifier.	
f	is a format specifier.	
keyspec	is a key specifier (see Section 7.2.1.5).	
kn	is a key-of-reference specifier.	
s	is the label of an executable statement.	
list	is an I/O list.	
	Reads one or more logical records, specified by key value, and assigns values to the elements in the list.	

Form	Effect	Manual Section

READ ([UNIT=]c,FMT=f,[,IOSTAT=ios][,ERR=s][,END=s])[list] 7.4.4

c is an internal file specifier.

f is a format specifier.

ios is an input/output status specifier.

s is the label of an executable statement.

list is an I/O list.

Reads into elements in the list one or more internal records containing character strings, converting in accordance with the format specification.

RETURN [(i)] 4.7

Returns control to the calling program from the current subprogram. The optional argument is an integer value that indicates which alternate return is to be taken.

REWIND([UNIT=]u[,IOSTAT=ios][,ERR=s]) 9.4
REWIND u

u is a logical unit specifier.

ios is an input/output status specifier.

s is the label of an executable statement.

Repositions logical unit u to the beginning of the currently opened file.

REWRITE([UNIT=]u,[FMT=]f[,IOSTAT=ios][,ERR=s])[list] 7.6

REWRITE([UNIT=]u[,IOSTAT=ios][,ERR=s])[list] 7.6

u is a logical unit specifier.

f is a format specifier.

ios is an input/output status specifier.

s is the label of an executable statement.

list is an I/O list.

Transfers data from internal storage to the current record in an indexed file.

Form	Effect	Manual Section

SAVE[a[,a]...] 5.6

a — is the name of a variable, an array, or a named common block enclosed in slashes.

Retains the definition status of an entity after the execution of a RETURN or END statement in a subprogram.

Statement Function 6.2.1

f([p[,p]...])=e

f — is a symbolic name.

p — is a symbolic name.

e — is an expression.

Creates a user-defined function having the variables p as dummy arguments. When referred to, the expression is evaluated using the actual arguments in the function call.

STOP [disp] 4.9

disp — is a decimal digit string containing 1 to 5 digits or a character constant.

Terminates program execution and prints the display, if one is specified.

SUBROUTINE nam[([p[,p]...])] 6.2.3

nam — is a symbolic name.

p — is a symbolic name or an alternate return specifier (*).

Begins a subroutine subprogram, indicating the program name and any dummy argument names (p).

TYPE — See WRITE 7.8

Form	Effect	Manual Section

Type Declaration 5.2

```
typ v[/clist/][,v[/clist/]...
```

typ is one of the following data type specifiers:

```
BYTE
LOGICAL
LOGICAL*1
LOGICAL*2
LOGICAL*4
INTEGER
INTEGER*2
INTEGER*4
REAL
REAL*4
REAL*8
REAL*16
DOUBLE PRECISION
COMPLEX
COMPLEX*8
COMPLEX*16
DOUBLE COMPLEX
CHARACTER*len
CHARACTER*(*)
```

v is a variable name, array name, function or function entry name, or an array declarator. The name can optionally be followed by a data type length specifier (*n). For character entities, the length specifier can be *len or *(*).

clist is an initial value or values to be assigned to the immediately preceding variable or array element.

The symbolic names (v) are assigned the specified data type.

```
UNLOCK ([UNIT=]u[,IOSTAT=ios][,ERR=s])
```
9.8

```
UNLOCK u
```
9.8

u is a logical unit specifier.

ios is an input/output specifier.

s is the label of an executable statement.

Removes the access protection from the file connected to Logical Unit u.

```
VIRTUAL a(d)[,a(d)]...
```
5.3

Equivalent to the DIMENSION statement.

Form	Effect	Manual Section
WRITE ([UNIT=]u,[FMT=]f[,IOSTAT=ios][,ERR=s])[list]		7.5.1.1
PRINT f[,list]		7.8
TYPE f[,list]		7.8

u is a logical unit specifier.

f is a format specifier.

ios is an input/output specifier.

s is a label of an executable statement.

list is an I/O list.

Writes one or more logical records to unit u, containing the values of the elements in the list. The records are converted according to f.

WRITE([UNIT=]u,[FMT=]*[,IOSTAT=ios][,ERR=s])[list]		7.5.1.2
WRITE(u,*[,ERR=s])[list]		7.5.1.2
PRINT *[,list]		7.8
TYPE *[,list]		7.8

u is a logical unit specifier.

* denotes list-directed formatting.

ios is an input/output specifier.

s is a label of an executable statement.

list is an I/O list.

Writes one or more logical records to unit u containing the values of the elements in the list. The records are converted according to the data type of the list element.

WRITE ([UNIT=]u[,IOSTAT=ios][,ERR=s])[list]		7.5.1.3

u is a logical unit specifier.

s is a label of an executable statement label.

ios is an input/output specifier.

list is an I/O list.

Writes one unformatted record to unit u containing the values of the elements in the list.

Form	Effect	Manual Section

WRITE ([UNIT=]u, REC=r, FMT=f[,IOSTAT=ios][,ERR=s])[list] 7.5.2.1

WRITE (u'r,f[,ERR=s])[list] 7.5.2.1

u is a logical unit specifier.

r is a relative record number specifier.

f is a format specifier.

ios is an input/output specifier.

s is a label of an executable statement.

list is an I/O list.

Writes the values of the elements of the list to record r on unit u. The record is converted according to f.

WRITE([UNIT=]u,REC=r[,IOSTAT=ios][,ERR=s])[list] 7.5.2.2

WRITE (u'r[,ERR=s]) [list] 7.5.2.2

u is a logical unit specifier.

r is a relative record number specifier.

ios is an input/output specifier.

s is a label of an executable statement label.

list is an I/O list.

Writes record r to unit u containing the values of the elements in the list.

WRITE ([UNIT=]c,FMT=f[,IOSTAT=ios][,ERR=s])[List] 7.5.4

c is an internal file specifier.

f is a format specifier.

s is the label of an executable statement.

ios is an input/output specifier.

list is an I/O list.

Writes elements in the list to the internal file specified by the unit, converting to character strings in accordance with the format specification.

C.3 LIBRARY FUNCTIONS

Table C-1 lists the VAX-11 FORTRAN generic functions and intrinsic functions (listed in the column headed "Specific Name"). Superscripts in the table refer to the notes that follow the table.

FORTRAN LANGUAGE SUMMARY

Table C-1 Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Square Root[1]	1	SQRT	SQRT	REAL*4	REAL*4
			DSQRT	REAL*8	REAL*8
			QSQRT	REAL*16	REAL*16
$a^{1/2}$			CSQRT	COMPLEX*8	COMPLEX*8
			CDSQRT	COMPLEX*16	COMPLEX*16
Natural Logarithm[2]	1	LOG	ALOG	REAL*4	REAL*4
			DLOG	REAL*8	REAL*8
			QLOG	REAL*16	REAL*16
$\log_e a$			CLOG	COMPLEX*8	COMPLEX*8
			CDLOG	COMPLEX*16	COMPLEX*16
Common Logarithm[2]	1	LOG10	ALOG10	REAL*4	REAL*4
			DLOG10	REAL*8	REAL*8
$\log_{10} a$			QLOG10	REAL*16	REAL*16
Exponential	1	EXP	EXP	REAL*4	REAL*4
e^a			DEXP	REAL*8	REAL*8
			QEXP	REAL*16	REAL*16
			CEXP	COMPLEX*8	COMPLEX*8
			CDEXP	COMPLEX*16	COMPLEX*16
Sine[3]	1	SIN	SIN	REAL*4	REAL*4
			DSIN	REAL*8	REAL*8
			QSIN	REAL*16	REAL*16
Sin a			CSIN	COMPLEX*8	COMPLEX*8
			CDSIN	COMPLEX*16	COMPLEX*16
Cosine[3]	1	COS	COS	REAL*4	REAL*4
			DCOS	REAL*8	REAL*8
			QCOS	REAL*16	REAL*16
Cos a			CCOS	COMPLEX*8	COMPLEX*8
			CDCOS	COMPLEX*16	COMPLEX*16
Tangent[3]	1	TAN	TAN	REAL*4	REAL*4
			DTAN	REAL*8	REAL*8
Tan a			QTAN	REAL*16	REAL*16
Arc Sine[4,5]	1	ASIN	ASIN	REAL*4	REAL*4
			DASIN	REAL*8	REAL*8
Arc Sin a			QASIN	REAL*16	REAL*16
Arc Cosine[4,5]	1	ACOS	ACOS	REAL*4	REAL*4
			DACOS	REAL*8	REAL*8
Arc Cos a			QACOS	REAL*16	REAL*16
Arc Tangent[4]	1	ATAN	ATAN	REAL*4	REAL*4
			DATAN	REAL*8	REAL*8
Arc Tan a			QATAN	REAL*16	REAL*16
Arc Tangent[5,6]	2	ATAN2	ATAN2	REAL*4	REAL*4
			DATAN2	REAL*8	REAL*8
Arc Tan a_1/a_2			QATAN2	REAL*16	REAL*16

Table C-1 (Cont.) Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Hyperbolic Sine	1	SINH	SINH	REAL*4	REAL*4
			DSINH	REAL*8	REAL*8
Sinh a			QSINH	REAL*16	REAL*16
Hyperbolic Cosine	1	COSH	COSH	REAL*4	REAL*4
			DCOSH	REAL*8	REAL*8
Cosh a			QCOSH	REAL*16	REAL*16
Hyperbolic Tangent	1	TANH	TANH	REAL*4	REAL*4
			DTANH	REAL*8	REAL*8
Tanh a			QTANH	REAL*16	REAL*16
Absolute value[7]	1	ABS	HABS	INTEGER*2	INTEGER*2
|a|			JIABS	INTEGER*4	INTEGER*4
			ABS	REAL*4	REAL*4
			DABS	REAL*8	REAL*8
			QABS	REAL*16	REAL*16
			CABS	COMPLEX*8	REAL*4
			CDABS	COMPLEX*16	REAL*8
		IABS	HABS	INTEGER*2	INTEGER*2
			JIABS	INTEGER*4	INTEGER*4
Truncation[8,11]	1	INT	HINT	REAL*4	INTEGER*2
[a]			JINT	REAL*4	INTEGER*4
			HIDINT	REAL*8	INTEGER*2
			JIDINT	REAL*8	INTEGER*4
			HIQINT	REAL*16	INTEGER*2
			JIQINT	REAL*16	INTEGER*4
			—	COMPLEX*8	INTEGER*2
			—	COMPLEX*8	INTEGER*4
			—	COMPLEX*16	INTEGER*2
			—	COMPLEX*16	INTEGER*4
		IDINT	HIDINT	REAL*8	INTEGER*2
			JIDINT	REAL*8	INTEGER*4
		IQINT	HIQINT	REAL*16	INTEGER*2
			JIQINT	REAL*16	INTEGER*4
		AINT	AINT	REAL*4	REAL*4
			DINT	REAL*8	REAL*8
			QINT	REAL*16	REAL*16
Nearest Integer[8,11]	1	NINT	ININT	REAL*4	INTEGER*2
[a+.5*sign(a)]			JNINT	REAL*4	INTEGER*4
			HIDNNT	REAL*8	INTEGER*2
			JIDNNT	REAL*8	INTEGER*4
			HIQNNT	REAL*16	INTEGER*2
			JIQNNT	REAL*16	INTEGER*4
		IDNINT	HIDNNT	REAL*8	INTEGER*2
			JIDNNT	REAL*8	INTEGER*4
		IQNINT	HIQNNT	REAL*16	INTEGER*2
			JIQNNT	REAL*16	INTEGER*4
		ANINT	ANINT	REAL*4	REAL*4
			DNINT	REAL*8	REAL*8
			QNINT	REAL*16	REAL*16

Table C-1 (Cont.) Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Conversion to[9] REAL*4	1	REAL	FLOATI	INTEGER*2	REAL*4
			FLOATJ	INTEGER*4	REAL*4
			—	REAL*4	REAL*4
			SNGL	REAL*8	REAL*4
			SNGLQ	REAL*16	REAL*4
			—	COMPLEX*8	REAL*4
			—	COMPLEX*16	REAL*4
Conversion to[6] REAL*8	1	DBLE	—	INTEGER*2	REAL*8
			—	INTEGER*4	REAL*8
			DBLE	REAL*4	REAL*8
			—	REAL*8	REAL*8
			DBLEQ	REAL*16	REAL*8
			—	COMPLEX*8	REAL*8
			—	COMPLEX*16	REAL*8
Conversion to REAL*16	1	QEXT	—	INTEGER*2	REAL*16
			—	INTEGER*4	REAL*16
			QEXT	REAL*4	REAL*16
			QEXTD	REAL*8	REAL*16
			—	REAL*16	REAL*16
			—	COMPLEX*8	REAL*16
			—	COMPLEX*16	REAL*16
Fix[8,11]	1	IFIX	IIFIX	REAL*4	INTEGER*2
			JIFIX	REAL*4	INTEGER*4
(REAL*4-to-integer conversion)					
Float[9]	1	FLOAT	FLOATI	INTEGER*2	REAL*4
			FLOATJ	INTEGER*4	REAL*4
(integer-to-REAL*4 conversion)					
REAL*8 Float[8]	1	DFLOAT	DFLOTI	INTEGER*2	REAL*8
			DFLOTJ	INTEGER*4	REAL*8
(integer-to-REAL*8 conversion)					
REAL*16 Float	1	QFLOAT	—	INTEGER*2	REAL*16
				INTEGER*4	REAL*16
(Integer to REAL*16 conversion)					
Conversion to COMPLEX*8,	1,2[12]	CMPLX	—	INTEGER*2	COMPLEX*8
or	1,2		—	INTEGER*4	COMPLEX*8
COMPLEX*8 from Two	1,2		—	REAL*4	COMPLEX*8
Arguments	1,2		—	REAL*8	COMPLEX*8
	1,2		—	REAL*16	COMPLEX*8
	1		—	COMPLEX*8	COMPLEX*8
	1		—	COMPLEX*16	COMPLEX*8

Table C-1 (Cont.) Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Conversion to COMPLEX*16, or COMPLEX*16 from Two Arguments	1,2[17]	DCMPLX	—	INTEGER*2	COMPLEX*16
	1,2		—	INTEGER*4	COMPLEX*16
	1,2		—	REAL*4	COMPLEX*16
	1,2		—	REAL*8	COMPLEX*16
	1,2		—	REAL*16	COMPLEX*16
	1		—	COMPLEX*8	COMPLEX*16
	1		—	COMPLEX*16	COMPLEX*16
Real Part of Complex	1	—	REAL	COMPLEX*8	REAL*4
			DREAL	COMPLEX*16	REAL*8
Imaginary Part of Complex	1	—	AIMAG	COMPLEX*8	REAL*4
			DIMAG	COMPLEX*16	REAL*8
Complex From Two Arguments	(See Conversion to COMPLEX*8 and Conversion to COMPLEX*16)				
Complex Conjugate	1	CONJG	CONJG	COMPLEX*8	COMPLEX*8
			DCONJG	COMPLEX*16	COMPLEX*16
(if a=(X,Y) CONJG (a)=(X,-Y)					
REAL*8 product of REAL*4's a_1*a_2	2	—	DPROD	REAL*4	REAL*8
Maximum[1]	n	MAX	IMAX0	INTEGER*2	INTEGER*2
			JMAX0	INTEGER*4	INTEGER*4
$\max(a_1,a_2,\ldots a_n)$			AMAX1	REAL*4	REAL*4
(returns the maximum value from among the			DMAX1	REAL*8	REAL*8
argument list; there must be at			QMAX1	REAL*16	REAL*16
least two arguments)		MAX0	IMAX0	INTEGER*2	INTEGER*2
			JMAX0	INTEGER*4	INTEGER*4
		MAX1	IMAX1	REAL*4	INTEGER*2
			JMAX1	REAL*4	INTEGER*4
		AMAX0	AIMAX0	INTEGER*2	REAL*4
			AJMAX0	INTEGER*4	REAL*4
Minimum[1]	n	MIN	IMIN0	INTEGER*2	INTEGER*2
			JMIN0	INTEGER*4	INTEGER*4
$\min(a_1,a_2,\ldots a_n)$			AMIN1	REAL*4	REAL*4
(returns the minimum value among the argument list;			DMIN1	REAL*8	REAL*8
there must be at least two			QMIN1	REAL*16	REAL*16
arguments)		MIN0	IMIN0	INTEGER*2	INTEGER*2
			JMIN0	INTEGER*4	INTEGER*4
		MIN1	IMIN1	REAL*4	INTEGER*2
			JMIN1	REAL*4	INTEGER*4
		AMIN0	AIMIN0	INTEGER*2	REAL*4
			AJMIN0	INTEGER*4	REAL*4

Table C-1 (Cont.) Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Positive Difference $a_1-(\min(a_1,a_2))$ (returns the first argument minus the minimum of the two arguments)	2	DIM	IIDIM	INTEGER*2	INTEGER*2
			JIDIM	INTEGER*4	INTEGER*4
			DIM	REAL*4	REAL*4
			DDIM	REAL*8	REAL*8
			QDIM	REAL*16	REAL*16
		IDIM	JIDIM	INTEGER*2	INTEGER*2
			JIDIM	INTEGER*4	INTEGER*4
Remainder $a_1-a_2*\lfloor a_1/a_2 \rfloor$ returns the remainder when the first argument is divided by the second)	2	MOD	IMOD	INTEGER*2	INTEGER*2
			JMOD	INTEGER*4	INTEGER*4
			AMOD	REAL*4	REAL*4
			DMOD	REAL*8	REAL*8
			QMOD	REAL*16	REAL*16
Transfer of Sign $\lvert a_1 \rvert * \text{Sign } a_2$	2	SIGN	IISIGN	INTEGER*2	INTEGER*2
			JISIGN	INTEGER*4	INTEGER*4
			SIGN	REAL*4	REAL*4
			DSIGN	REAL*8	REAL*8
			QSIGN	REAL*16	REAL*16
		ISIGN	IISIGN	INTEGER*2	INTEGER*2
			JISIGN	INTEGER*4	INTEGER*4
Bitwise AND (performs a logical AND on corresponding bits)	2	IAND	IIAND	INTEGER*2	INTEGER*2
			JIAND	INTEGER*4	INTEGER*4
Bitwise OR (performs an inclusive OR on corresponding bits)	2	IOR	IIOR	INTEGER*2	INTEGER*2
			JIOR	INTEGER*4	INTEGER*4
Bitwise Exclusive OR (performs an exclusive OR on corresponding bits)	2	IEOR	IIEOR	INTEGER*2	INTEGER*2
			JIEOR	INTEGER*4	INTEGER*4
Bitwise Complement (complements each bit)	1	NOT	INOT	INTEGER*2	INTEGER*2
			JNOT	INTEGER*4	INTEGER*4
Bitwise Shift (a_1 logically shifted left a_2 bits)	2	ISHFT	IISHFT	INTEGER*2	INTEGER*2
			JISHFT	INTEGER*4	INTEGER*4
Length[1] (returns length of the character expression)	1	—	LEN	CHARACTER	INTEGER*4
Index (C_1, C_2)[1] (returns the position of the substring c_2 in the character expression c_1)	2	—	INDEX	CHARACTER	INTEGER*4
Character[1] (returns a character that has the ASCII value specified by the argument)	1	—	CHAR	LOGICAL*1	CHARACTER
				INTEGER*2	
				INTEGER*4	

Table C-1 (Cont.) Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
ASCII Value[10] (returns the ASCII value of the argument; the argument must be a character expression that has a length of 1)	1	—	ICHAR	CHARACTER	INTEGER*4
Character relationals (ASCII collating sequence)	2	—	LLT	CHARACTER	LOGICAL*4
	2	—	LLE	CHARACTER	LOGICAL*4
	2	—	LGT	CHARACTER	LOGICAL*4
	2	—	LGE	CHARACTER	LOGICAL*4

Notes for Table C-1

[1] The argument of SQRT, DSQRT, and QSQRT must be greater than or equal to zero. The result of CSQRT or CDSQRT is the principal value with the real part greater than or equal to zero. When the real part is zero, the result is the principal value with the imaginary part greater than or equal to zero.

[2] The argument of ALOG, DLOG, QSQRT, ALOG10, DLOG10, and QLOG10 must be greater than zero. The argument of CLOG or CDLOG must not be (0.,0.).

[3] The argument of SIN, DSIN, QSIN, COS, DCOS, QCOS, TAN, DTAN, and QTAN must be in radians. The argument is treated modulo 2*pi.

[4] The absolute value of the argument of ASIN, DASIN, QASIN, ACOS, DACOS and QACOS must be less than or equal to 1.

[5] The result of ASIN, DASIN, QASIN, ACOS, DACOS, QACOS, ATAN, DATAN, QATAN, ATAN2, DATAN2, and QATAN2 is in radians.

[6] The result of ATAN2, DATAN2, and QATAN2 is zero or positive when a_y is less than or equal to zero. The result is undefined if both arguments are zero.

[7] The absolute value of a complex number, (X,Y), is the real value:

$$(X^2+Y^2)^{1/2}$$

[8] [x] is defined as the largest integer whose magnitude does not exceed the magnitude of x and whose sign is the same as that of x. For example [5.7] equals 5. and [-5.7] equals -5.

[9] Functions that cause conversion of one data type to another type provide the same effect as the implied conversion in assignment statements. The function REAL with a real argument, the function DBLE with a double precision argument, the function INT with an integer argument, and the function QEXT with a REAL*16 argument return the value of the argument without conversion.

[10] See Chapter 6 for additional information on character functions.

[11] The functions INT, IDINT, IQINT, NINT, IDNINT, IQNINT, IFIX, MAX1, and MIN1 return INTEGER*4 values if the /I4 command qualifier is in effect, INTEGER*2 values if the /NOI4 qualifier is in effect.

[12] When CMPLX and DCMPLX have only one argument, this argument is converted into the real part of a complex value, and zero is assigned to the imaginary part; when there are two arguments (not complex), a complex value is produced by conversion of the first argument into the real part of the value, the second argument into the imaginary part.

C.4 SYSTEM SUBROUTINE SUMMARY

The VAX-11 FORTRAN system provides subroutines you call in the same manner as a user-written subroutine. These subroutines are described in this section.

The subroutines supplied are:

DATE	Returns a 9-byte string containing the ASCII representation of the current date.
IDATE	Returns three integer values representing the current month, day, and year.
ERRSNS	Returns information about the most recently detected error condition.
EXIT	Terminates the execution of a program and returns control to the operating system.
SECNDS	Provides system time of day, or elapsed time, as a floating-point value in seconds.
TIME	Returns an 8-byte string containing the ASCII representation of the current time in hours, minutes, and seconds.
RAN	Returns the next number from a sequence of pseudo-random numbers of uniform distribution over the range 0 to 1.

References to integer arguments in the following subroutine descriptions refer to arguments of either INTEGER*4 data type or INTEGER*2 data type. However, the arguments must be either all INTEGER*4 or all INTEGER*2. In general, INTEGER*4 variables or array elements may be used as input values to these subroutines if their value is within the INTEGER*2 range.

C.4.1 DATE

The DATE subroutine obtains the current date as set within the system. The call to DATE has the form

```
CALL DATE(buf)
```

where:

buf
is a 9-byte variable, array, array element, or character substring. The date is returned as a 9-byte ASCII character string of the form

```
dd-mmm-yy
```

where:

dd
is the 2-digit date.

mmm
is the 3-letter month specification.

yy
is the last two digits of the year.

C.4.2 IDATE

The IDATE subroutine returns three integer values representing the current month, day, and year. The call to IDATE has the form

```
CALL IDATE(i,j,k)
```

If the current date were October 9, 1984, the values of the integer variables upon return would be:

```
i = 10
j = 9
k = 84
```

C.4.3 ERRSNS

The ERRSNS subroutine returns information about the most recent error that has occurred during program execution. The call to ERRSNS has the form

```
CALL ERRSNS(fnum,rmssts,rmsstv,iunit,condval)
```

where:

fnum
is an integer variable or array element into which is stored the most recent FORTRAN error number. VAX-11 FORTRAN error numbers are listed in the VAX-11 FORTRAN User's Guide.

A zero is returned if no error has occurred since the last call to ERRSNS, or if no error has occurred since the start of execution.

rmssts
is an integer variable or array element into which is stored the RMS completion status code (STS), if the last error was an RMS I/O error.

rmsstv
is an integer variable or array element into which is stored the RMS status value (STV) if the last error was an RMS I/O error. This status value provides additional status information.

iunit
is an integer variable or array element into which is stored the logical unit number, if the last error was an I/O error.

condval
is an integer variable or array element into which is stored the actual VAX-11 condition value.

Any of the arguments may be null. If the arguments are of INTEGER*2 type, only the low-order 16 bits of information are returned. The saved error information is set to zero after each call to ERRSNS.

C.4.4 EXIT

The EXIT subroutine causes program termination, closes all files, and returns control to the operating system. A call to EXIT has the form

```
CALL EXIT [(exit-status)]
```

where:

(exit-status)

is an optional integer argument you can use to specify the image exit-status value.

C.4.5 SECNDS

The SECNDS function subprogram returns the system time in seconds as a single-precision, floating-point value less the value of its single-precision, floating-point argument. The call to SECNDS has the form

```
y = SECNDS(x)
```

where:

y

is set equal to the time in seconds since midnight, minus the user-supplied value of x.

The SECNDS function can be used to perform elapsed-time computations. For example:

```
C     START OF TIMED SEQUENCE
      T1 = SECNDS(0.0)
C
C     CODE TO BE TIMED
C
      DELTA = SECNDS(T1)
```

where:

DELTA

will give the elapsed time.

The value of SECNDS is accurate to 0.01 seconds, which is the resolution of the system clock.

NOTES

1. The time is computed from midnight. SECNDS also produces correct results for time intervals that span midnight.

2. The 24 bits of precision provides accuracy to the resolution of the system clock for about one day. However, loss of significance can occur if you attempt to compute very small elapsed times late in the day. More precise timing information can be obtained using Run Time Library procedures:

```
LIB$INIT_TIMER
LIB$SHOW_TIMER
LIB$STAT_TIMER
```

C.4.6 TIME

The TIME subroutine returns the current system time as an ASCII string. The call to TIME has the form

```
CALL TIME(buf)
```

where buf is an 8-byte variable, array, array element, or character substring.

The TIME call returns the time as an 8-byte ASCII character string of the form

```
hh:mm:ss
```

where:

hh
is the 2-digit hour indication.

mm
is the 2-digit minute indication.

ss
is the 2-digit second indication.

For example:

```
10:45:23
```

A 24-hour clock is used.

C.4.7 RAN

The RAN function is a general random number generator of the multiplicative congruential type. The result is a floating-point number that is uniformly distributed in the range between 0.0 inclusive and 1.0 exclusive. The call to RAN has the form:

```
y=RAN(i)
```

where:

y

is set equal to the value associated, by the function, with the argument i. The argument i must be an INTEGER*4 variable or INTEGER*4 array element.

The argument should initially be set to a large, odd integer value. The RAN function stores a value in the argument that it later uses to calculate the next random number.

There are no restrictions on the seed, although it should be initialized with different values on separate runs in order to obtain different random numbers; the seed is updated automatically. RAN uses the following algorithm to update the seed passed as the parameter:

```
SEED = 69069 * SEED + 1 (MOD 2**32)
```

The value of SEED is a 32-bit number whose high-order 24 bits are converted to floating point and returned as the result.

Appendix A
VAX-11 BASIC Language Elements

This appendix summarizes the program elements, commands, statements, operators, and functions supported by BASIC. More information on language elements is available in the *VAX-11 BASIC Language Reference Manual*.

A.1 Program Elements

BASIC programs can contain:

1. Arrays

   An array is an ordered arrangement of elements (subscripted variables) in one or two dimensions. You specify an array with a real, integer, or string variable name followed by integer subscripts in the range 0 to 32767. Enclose subscripts in parentheses. Noninteger subscripts are truncated to an integer value. A single subscript indicates a one-dimensional array, or list; two subscripts separated by commas indicate a two-dimensional array, or matrix.

   You should explicitly initialize all variables in virtual arrays at the start of your program.

2. Backslash Statement Separators

   The backslash statement separator (\) separates statements in a multi-statement line.

3. Character Set

   BASIC uses the full ASCII character set. This set includes:

   a. The letters A through Z, in both upper and lower case

   b. The numbers 0 through 9

   c. Special characters

Appendix D contains the full ASCII character set and each character's value. The compiler:

a. Interprets a lower-case letter to be the same as the corresponding upper-case letter, except in string literals

b. Does not process nonprinting characters (unless they are part of a string literal)

c. Does not process characters in REMARK statements or comment fields

4. Comments

Comments begin with an exclamation point (!) and end with a line terminator. You can insert comments after any statement except the DATA statement.

5. Constants

BASIC accepts three types of constants: real, integer, and string. Real constants are decimal numbers in the range $1E^{-38}$ to $1E^{38}$. Integer constants are whole numbers in the range -32768 to 32767 for programs compiled with the /WORD qualifier, and -2147483647 to 2147483647 for programs compiled with the /LONG qualifier.

String constants are alphanumeric characters delimited by matched pairs of single or double quotation marks. Quoted strings can contain from zero to 65535 characters.

6. Continued Lines

Program lines continue to the next text line if they end with an ampersand (&) immediately followed by a line terminator.

7. Data Types

VAX-11 BASIC accepts five data types:

a. SINGLE (32-bit floating-point numbers)

b. DOUBLE (64-bit floating-point numbers)

c. WORD (16-bit integers)

e. LONG (32-bit integers)

f. STRING (8-bit bytes (characters))

You select the type of integer or real data by specifying a qualifier when you compile your program. You can also declare a variable's data type explicitly with DECLARE, DIM, DEF, MAP, COMMON, and SUB statements.

8. Expressions

Expressions contain constants, variables, or functions, separated by operators.

9. Functions

Functions perform a series of numeric or string operations on the arguments you specify and return the result to your program. Functions are multi-character names followed by optional parentheses. The parentheses contain one to eight function arguments separated by commas. A null argument is not allowed. User-defined functions follow this general format; however, the function name begins with FN followed by 1 to 28 letters, digits, or periods. If a user-defined function name ends with a dollar sign, it returns string values. If it ends with a percent sign, it returns integer values. Otherwise, it returns real values. You can also write function subprograms. See the *VAX-11 BASIC User's Guide* for more information.

10. Line Length

A program line can contain any number of text lines. However, each line of text cannot exceed 256 characters.

11. Line Numbers

All program lines except continuation lines need line numbers. BASIC line numbers are positive whole numbers in the range 1 to 32767, inclusive. Numbers outside this range, fractional line numbers, line numbers with embedded spaces or tabs, and line numbers with percent signs generate errors. Leading zeros are ignored.

12. Line Terminators

A carriage return/line feed sequence (RET) ends a program line.

13. Operators

BASIC accepts arithmetic, relational, and logical operators. See Tables A-1 through A-3 at the end of this appendix.

14. Variables

BASIC accepts three types of variables: floating-point, integer, and string.

a. Floating-point variables contain a single letter, followed by up to 29 optional letters, underscores, dollar signs, digits, and periods.

b. Integer variables contain a single letter, followed by up to 28 optional letters, underscores, dollar signs, digits, and periods, followed by a percent sign.

c. String variables contain a single letter, followed by up to 28 optional letters, underscores, dollar signs, digits, and periods, followed by a dollar sign.

With the DECLARE, MAP, COMMON and DIM statements, you can explicitly assign a data type to a variable. Variable names in these statements cannot end in a percent or dollar sign.

You can use any combination of alphanumeric characters except keywords for a variable name. Using keywords generates compilation errors. Unless

they are part of a MAP, COMMON, or virtual array, variables are initialized to zero or a null string at the start of program execution.

Variable names cannot start with FN.

A.2 Commands

Commands allow you to perform operations on your program. They do not need line numbers. You can type them directly to BASIC along with any valid arguments.

The following is a short description of the BASIC commands, including their format and use. See the *VAX-11 BASIC User's Guide* for more information.

BASIC Compiler Commands

APPEND	merges the specified program with the program currently in memory.
COMPILE	generates an object module from a BASIC source program. The object module has a default file type of .OBJ.
CONTINUE	resumes execution after a STOP statement or a CTRL/C.
DELETE	erases a specified line(s) from the current BASIC source program in memory.
EDIT	changes source text within a line.
EXIT	exits BASIC to DCL command level.
HELP	prints help information on the terminal.
IDENTIFY	causes BASIC to print an identification header on the terminal.
INQUIRE	identical to the HELP command.
LIST	prints the current source program on the terminal.
LISTNH	prints the current source program on the terminal without header information.
LOAD	loads an object module into memory.
LOCK	identical to the SET command.
NEW	clears memory for the creation of a new program.
OLD	reads a specified BASIC source program into memory.
RENAME	changes the name of the program currently in memory.
REPLACE	saves a new version of a BASIC source program on disk.
RUN	prints header information and executes the program currently in memory. This program can be either: (1) a BASIC source program placed in memory with the OLD command, or (2) an object module placed in memory with the LOAD command.

RUNNH	is the same as RUN command, but does not print header information.
SAVE	writes a source program to a specified device.
SCALE	controls accumulated round-off errors for numeric operations.
SEQUENCE	generates line numbers for input text.
SET	specifies default compiler qualifiers. SET is identical to LOCK.
SCRATCH	clears memory.
SHOW	displays current default compiler qualifiers.
UNSAVE	deletes a specified file.

A.3 Statements

Statements assign values, perform I/O, transfer program control, and so forth. Each statement in the following section includes a description of what the statement does, a meta-language representation of format, and one or more sample program lines.

CALL

The CALL statement transfers control to a subprogram, optionally passes parameters to it, and stores the location of the calling program for an eventual return.

Format

```
CALL sub-nam [BY REF | BY DESC | BY VALUE] (exp [BY REF | BY DESC | BY VALUE] , [exp [BY REF | BY DESC | BY VALUE] ...])]
```

Example

```
200 CALL SUB1 BY REF (A$, Z BY VALUE, D$() BY DESC)
```

CHAIN

The CHAIN statement transfers control from the current program to another BASIC program. The program to which you CHAIN must be in executable format.

Format

```
CHAIN str-exp
```

Example

```
240 CHAIN "COSINE.EXE"
```

CHANGE

The CHANGE statement (1) converts a string of characters to their decimal ASCII values (Format 1), or (2) converts a list of numbers into a string of ASCII characters (Format 2).

Format 1

```
CHANGE str-exp TO num-array
```

Example 1

```
50  DIM ARRAY.CHANGES(6)
60  CHANGE A$ TO ARRAY.CHANGES
```

Format 2

```
CHANGE num-array TO str-vbl
```

Example 2

```
200 CHANGE ARRAY.CHANGES TO A$
```

CLOSE

The CLOSE statement ends I/O processing to a device or file.

Format

```
CLOSE #chnl-exp [,chnl-exp]. . .
```

Example

```
15000 CLOSE #5%
```

COMMON

The COMMON statement defines a named, shared area of storage called a COMMON block. This block stores values that are available to be read by any other program module. The COMMON name is one to thirty characters long.

Format

```
COM[MON] [(com-nam)]  [data-type]  { num-vbl                                  }[,...]
                                   { str-vbl[=int-exp]                        }[,...]
                                   { num-arr(num-cnst[,num-cnst])             }[,...]
                                   { str-arr(num-cnst[,num-cnst])[=num-cnst]  }[,...]
                                   { FILL-item                                }[,...]
```

Example

```
500 COMMON SHELF_NUMBER%, ROW$=2%, PART_BIN
```

DATA

The DATA statement creates a data block for the READ statement. A DATA statement must be the only statement on a line, or the last statement on a multi-statement line. DATA items must be separated with commas.

Format

```
DATA { num-cnst
     { str-cnst
     { unquoted-string } [,...]
     { q-str
```

Example

```
30 DATA 35,32.3,PRODUCTION SEQUENCE,"SYSTEM","1.2"
```

DECLARE

The DECLARE statement explicitly assigns a data type to a variable, function, or constant. The datatype must be one of the following:

- WORD, specifying a 16-bit integer
- LONG, specifying a 32-bit integer
- INTEGER, defaulting to the integer data type specified at compile time
- REAL, defaulting to the floating-point data type specified at compile time
- STRING, specifying a string

Variables ending in a percent sign (%) or a dollar sign ($) are invalid. Variables named in a DECLARE statement cannot also be named in a COMMON, MAP, SUB, FUNCTION, or DIM statement.

Format 1

```
DECLARE data-type vbl [,vbl. . .]
```

Example 1

```
10 DECLARE INTEGER A, B
```

Format 2

```
DECLARE data-type CONSTANT cnst-nam = value
```

Example 2

```
200 DECLARE REAL CONSTANT E = 2.71828
```

Format 3

```
DECLARE data-type FUNCTION function-name
```

Example 3

```
2000 DECLARE REAL FUNCTION Z
2100 DEF Z(FIRST, SECOND) = FIRST / SECOND
```

DEF

The DEF statement defines a user-created function. Format 1 is a single-line function definition; Format 2 is a multi-line function definition.

Format 1

```
DEF FNvbl1 [([data-type] vbl2 [,[data-type] vbl3]. . .)] = exp
```

Example 1

```
10 DEF FN.SIGMA(A, B, C, D, E) = A + B + C + D + E
```

Format 2

```
DEF FNvbl1 [([data-type] vbl2 [,[data-type] vbl3]. . .)]
```

Example 2

```
10 DEF FN.SIGMA(A, B, C, D, E)
20 FN.SIGMA = A + B + C + D + E
30 FNEND
```

DELETE

The DELETE statement logically removes a record from a file.

Format

```
DELETE [#]chnl-exp
```

Example

```
1000 DELETE #5%
```

DIMENSION

The DIMENSION statement declares an array and specifies: (1) the number of dimensions, and (2) the maximum value of each subscript. Format 1 describes an array in memory, Format 2 describes a virtual array. Refer to Appendix H for more information about virtual arrays.

Format 1

```
DIM[ENSION] data-type array-nam(int-exp1 [,int-exp2]) [,array-nam. . .]
```

Example 1

```
30 DIM INTEGER.ARRAY%(10,10), LIST.REAL(100), STRING.ARRAY$(100,100)
```

Format 2

```
DIM[ENSION] #chnl-exp, array-nam(int-exp6[,int-exp7])[=int-exp8] [. . .]
```

Example 2

```
100 DIM #12, LIST$(500), R(10,10)
```

END

The END statement marks the physical and logical end of a program. It must have the highest line number in the main program. Execution of the END statement closes all open files and releases all program storage.

Format

```
END
```

Example

```
32767 END
```

EXTERNAL

The EXTERNAL statement enables you to access external program symbols, system services, and function subprograms. The data type must be one of the following:

- WORD, specifying a 16-bit integer
- LONG, specifying a 32-bit integer
- INTEGER, defaulting to the integer data type specified at compile time
- REAL, defaulting to the floating-point data type specified at compile time
- STRING, specifying a string (valid only with FUNCTION)

FUNCTION specifies that the symbol is accessed as a function. CONSTANT specifies that the symbol is a named constant. If you do not specify FUNCTION or CONSTANT, you access the symbol as if it were a variable. The symbol can be any valid program symbol of up to 30 characters. You pass parameters to an EXTERNAL function or system service exactly as you pass parameters in the CALL statement, including null parameters.

Format

```
EXTERNAL data-type [FUNCTION | CONSTANT] symbol
```

Example

```
100 EXTERNAL INTEGER FUNCTION SYS$CREMBX
200 SYS_STATUS% = SYS$CREMBX(,CHAN%,,,,,"LINK01")
```

FIND

The FIND statement locates a specified record. Format 1 finds the next logical record in a SEQUENTIAL, RELATIVE, or INDEXED file. Format 2 finds a specified record in a RELATIVE or VIRTUAL file. Format 3 finds a specified record in an INDEXED file.

Format 1

```
FIND #chnl-exp
```

Example 1

```
200 FIND #4%
```

Format 2

```
FIND #chnl-exp ,RECORD int-exp
```

Example 2

```
400 FIND #4%, RECORD 155%
```

Format 3

```
FIND #chnl-exp,KEY #int-exp {EQ} {str-exp}
                            {GE} {int-exp}
                            {GT}
```

Example 3

```
200 FIND #4%, KEY #0% GE "ADAMS"
```

FNEND

The FNEND statement marks the physical and logical end of a function definition. The FNEND statement must be at the end of a multi-line function definition.

Format

```
FNEND
```

Example

```
10 DEF FN.DIFFERENCE (A,B)
20 FN.DIFFERENCE = A - B
30 FNEND
```

FNEXIT

The FNEXIT statement causes the program to exit from a multi-line function. You can use the FNEXIT statement in a multi-line DEF only. It is equivalent to a branch to the multi-line DEF's FNEND statement.

Format

```
FNEXIT
```

Example

```
10 DEF FN.SQUARE.ROOT (A)
20 IF A < 0 THEN FNEXIT
30 FN.SQUARE.ROOT = SQR(A)
40 FNEND
```

FOR

The FOR statement enables you to construct program loops. Formats 1 and 2 describe a multi-line loop. Format 2 uses additional qualifiers; the loop executes as long as: (1) num-exp2 has not reached the terminating condition, and (2) the WHILE expression is true, or the UNTIL condition is false. In Format 3, FOR is a statement modifier. BASIC executes <stmnt> until the terminating condition is reached.

Format 1

```
FOR num-vbl = num-exp1 TO num-exp2 [STEP num-exp3]
```

Example 1

```
10 FOR I% = 1% TO 10% STEP 2%
.
.
.
100 NEXT I%
```

Format 2

```
FOR num-vbl = num-exp1 TO num-exp2 [STEP num-exp3] {WHILE / UNTIL} exp
```

Example 2

```
10 FOR N% = 5% TO 20% STEP 4% WHILE 400 - Z < .0001
.
.
.
300 NEXT N%
```

Format 3

```
stmnt FOR num-vbl = num-exp1 TO num-exp2 STEP num-exp3
```

Example 3

```
400 PRINT I% ** I% FOR I% = 1% TO 3%
```

FREE

The FREE statement unlocks all records or buckets for a specified channel. The file specified by <chnl-exp> must be open before you can execute the FREE statement. You cannot use the FREE statement on: (1) terminal-format files, or (2) nondisk files.

Format

```
FREE #chnl-exp
```

Example

```
450 FREE #6%
```

FUNCTION

The FUNCTION statement marks the beginning of a function subprogram and specifies its parameters by number and data type. You declare the function subprogram in an EXTERNAL statement, then invoke it as you would any BASIC function.

Format

```
FUNCTION data-type sub-nam [([data-type] vbl, [data-type] vbl . . .)]
```

Example

```
1000 FUNCTION REAL FACTOR (REAL A)
```

FUNCTIONEND

The FUNCTIONEND statement marks the end of a function subprogram and returns control to the invoking program.

Format

```
FUNCTIONEND
```

Example

```
2000 FUNCTIONEND
```

FUNCTIONEXIT

The FUNCTIONEXIT statement returns control to the invoking program.

Format

```
FUNCTIONEXIT
```

Example

```
700 IF A > 1000 THEN FUNCTIONEXIT
```

GET

The GET statement makes the next logical record available for processing. Format 1 reads the next logical record from SEQUENTIAL, RELATIVE or INDEXED files. Format 2 reads a specified record from a RELATIVE or VIRTUAL file. Format 3 reads a specified record from an INDEXED file.

Format 1

```
GET #chnl-exp
```

Example 1

```
100 GET #4%
```

Format 2

```
GET #chnl-exp, RECORD int-exp
```

Example 2

```
500 GET #4%, RECORD 33
```

Format 3

```
GET #chnl-exp  ,KEY #int-exp {GE|EQ|GT} {str-exp|int-exp}
```

Example 3

```
560 GET #4%, KEY 0% GT "LEWIS"
```

GOSUB

The GOSUB statement transfers control to a specified line number and stores the location to which the subroutine returns.

Format

```
GOSUB   fin-num
GO SUB
```

Example

```
200 GOSUB 1100
1100 REM SUBROUTINE
      .
      .
      .
2100 RETURN
```

GOTO

The GOTO statement transfers control to a specified line number. <Lin-num> must be a valid line number in the same program unit as the GOTO statement.

Format

```
GOTO
GO TO   lin-num
```

Example

```
20 GOTO 200
```

IF

The IF statement evaluates a conditional expression and transfers program control depending on the resulting value. Format 1 describes the IF-THEN-ELSE construction. In Format 2, IF is a statement modifier; BASIC executes <stmnt> if the conditional expression is true.

Format

Format 1

```
IF  rel-exp   THEN  lin-num          ELSE  lin-num
    log-exp         stmnt(s)               stmnt(s)
                    GOTO lin-num
```

Example

```
100 IF A > 0 THEN PRINT A ELSE PRINT "A IS NOT GREATER THAN 0"
```

Format 2

```
stmnt IF cond-exp
```

Example 2

```
100 PRINT A IF A > 0
```

INPUT

The INPUT statement assigns values from your terminal or a terminal-format file to program variables.

Format

```
INPUT [#chnl-exp,][str-cnst {,} ] vbl [,vbl]. . .
                            {;}
```

Examples

```
40 INPUT "TYPE IN 3 INTEGERS"(A%, B%, C%
```

or

```
100 INPUT #3%, RECORD.STRING$
```

INPUT LINE

The INPUT LINE statement assigns a string value (including the line terminator(s)) from a terminal or terminal-format file to a string variable.

Format

```
INPUT LINE [#chnl-exp,][str-cnst {;} ] str-vbl
          [{:}]
```

Examples

```
650 INPUT LINE "TYPE A LINE OF TEXT", Z$
```

or

```
390 INPUT LINE #4%, RECORD.STRING$
```

KILL

The KILL statement erases a file.

Format

```
KILL str-exp
```

Example

```
200 KILL "TEMP.DAT"
```

LET

The LET statement assigns a value to one or more variables.

Format

```
[LET] vbl [,vbl]. . . = exp
```

Examples

```
10 LET A = 3.1415926535
```

or

```
20 A$ = "ABCDEFG"
```

LINPUT

The LINPUT statement assigns a string value from a terminal or terminal-format file to a string variable.

Format

```
LINPUT [#chnl-exp,][str-cnst {;|:}] str-vbl
```

Example

```
30 LINPUT "ENTER YOUR LAST NAME";I LAST.NAME$
```

or

```
55 LINPUT #2%, EMPLOYEE.NUMBER$
```

LSET

The LSET statement assigns left-justified data to a string variable. LSET does not change the length of a string variable.

Format

```
LSET str-vbl [,str-vbl] = str-exp
```

Example

```
400 A$ = "A"
410 LSET A$ = "XYZ"
```

Because LSET does not change a string's length, line 30 assigns the value "X" to A$.

MAP

The MAP statement, like the COMMON statement, defines a named, shared area of storage. Additionally, it can declare data fields in a file's record buffer, and associate these fields with program variables.

Format

```
MAP (map-nam) [data-type] { num-vbl
                          | str-vbl[=int-exp]
                          | num-arr(num-cnst[,num-cnst])
                          | str-arr(num-cnst[,num-cnst])[=num-cnst]
                          | FILL-item } [...]
```

Example

```
200 MAP (BUF1) ELEV%, FILL%(4), AZIM(100), FILL(2), GROUP$(8)=10
```

MARGIN

The MARGIN statement specifies the margin width for records in a terminal-format file.

Format

```
MARGIN [#chnl-exp,] int-exp
```

Example

```
30 MARGIN #4%, 132%
```

MAT

The MAT statement enables you to:

1. Assign values to array elements
2. Redimension a previously dimensioned array
3. Perform matrix multiplication, addition and subtraction
4. Transpose or invert a matrix, and find the resulting determinant

Format 1

```
MAT arr-nam = {CON | IDN | NUL$ | ZER} [(subs-exp1 [,subs-exp2])]
```

Example 1

```
10 MAT Z$ = NUL$
```

Format 2

```
MAT arr-nam1 = arr-nam-2 [{* | - | .} arr-nam-3]
```

Example 2

```
500 MAT A = NEW_VALUE + OLD_VALUE
```

Format 3

```
MAT arr-nam-4 = (num-exp) * arr-nam-5
```

Example 3

```
200 MAT Z = (5.3) * A
```

Format 4

```
MAT arr-nam-6 = {INV | TRN} (arr-nam-7)
```

Example 4

```
300 MAT Q = TRN A
```

MAT INPUT

The MAT INPUT statement assigns values from a terminal or terminal-format file to array elements.

Format

```
MAT INPUT [#chnl-exp] arr-nam [(subs-exp1[,subs-exp2])] [,arr-nam. . .]
```

Example

```
100 MAT INPUT A$(4,4)
```

MAT LINPUT

The MAT LINPUT statement receives string data from a terminal or terminal-format file and assigns it to string array elements.

Format

```
MAT LINPUT [#chnl-exp,] str-arr-nam[(subs-exp1[,subs-exp2])] .
```

Example

```
400 MAT LINPUT TIME,CARD$(10%)
```

MAT PRINT

The MAT PRINT statement prints the contents of an array on your terminal or assigns the value of each array element to a record in a terminal-format file.

Format

```
MAT PRINT [#chnl-exp,] arr-nam [(subs-exp1[subs-exp2])][{,|;}] [,arr-nam. . .]
```

Example

```
MAT PRINT ARR1(5,5)
```

MAT READ

The MAT READ statement assigns values from DATA statements to array elements.

Format

```
MAT READ arr-nam[(subs-exp1[,subs-exp2])] [,arr-nam. . .]
```

Example

```
100 MAT READ ANIMAL$(2,2)
200 DATA CAT,DOG,GOAT,RABBIT
```

MOVE

The MOVE statement moves data in a record buffer to or from specified variables.

Format

```
MOVE {FROM} #chnl-exp {num-vbl                                          }[...]
     {TO  }           {str-vbl [= int-exp1]                             }[...]
                      {num-arr-nam (subs-exp1[,subs-exp2]) ..           }[...]
                      {str-arr-nam (subs-exp1[,subs-exp2]) [= int-exp2] }[...]
                      {FILL-item                                        }
```

Examples

```
990 MOVE FROM #4%, RZ, FILL$ = V%,,E%, RB1%, BA
```

or

```
1000 MOVE TO #9%, FILL$ = 10%, A$ = 0A%, B$ = 30%, C$ = 2%
```

NAME AS

The NAME AS statement changes the name of a specified file.

Format

```
NAME str-exp1 AS str-exp2
```

Example

```
400 NAME "OUT.DAT" AS "RERUN.DAT"
```

NOMARGIN

The NOMARGIN statement cancels the effect of MARGIN.

Format

```
NOMARGIN chnl-exp
```

Example

```
100 NOMARGIN #5%
```

ON ERROR GO BACK

After BASIC executes an ON ERROR GO BACK in a subprogram or DEF, control transfers to the calling program when an error occurs.

Format

```
{ON ERROR GO BACK}
{ONERROR GO BACK }
```

Example

```
5 ON ERROR GO BACK
```

ON ERROR GOTO

The ON ERROR GOTO statement specifies a line number in the current program unit to which BASIC transfers control when an error occurs.

Format

```
{ON ERROR GOTO } lin-num
{ONERROR GOTO } lin-num
```

Example

```
5 ON ERROR GOTO 9999
```

ON ERROR GOTO 0

The ON ERROR GOTO 0 statement disables user error handling, and returns to system error handling.

Format

```
ON ERROR GOTO 0
```

Example

```
19000 ON ERROR GOTO 0
```

ON GOSUB

The ON GOSUB statement transfers program control to one of several subroutines, depending on the value of a control expression.

Format

```
ON int-exp [GOSUB] lin-num [,lin-num]. . .
```

Example

```
150 ON CONTROL% GOSUB 100,200,300,400
```

ON GOTO

The ON GOTO statement transfers program control to one of several target line numbers, depending on the value of a control expression.

Format

```
ON int-exp GOTO lin-num [,lin-num]. . .
```

Example

```
330 ON INDEX% GOTO 700,800,900
```

OPEN

The OPEN statement opens a file for processing processing. It transfers user-specified file characteristics to the Record Management Services and verifies the results.

Format

```
OPEN str-exp1 [{FOR OUTPUT} ]AS FILE #chnl-exp1
              [{FOR INPUT  }]

[ [,[ORGANIZATION] {VIRTUAL    }                          ]
                   {UNDEFINED  }                          ]
                   {INDEXED    }                          ]
                   {SEQUENTIAL }                          ]
                                 {VARIABLE}               ]
                                 {RELATIVE}               ]
                                 {FIXED   }               ]

[ [, ALLOW      {NONE  }]]
                {READ  }]
                {WRITE }]
                {MODIFY}]

[ [, ACCESS     {READ  }]]
                {WRITE }]
                {MODIFY}]
                {SCRATCH}]
                {APPEND}]

[ [, RECORDTYPE {LIST   }]]
                {FORTRAN}]
                {NONE   }]
                {ANY    }]

[, RECORDSIZE int-exp1]

[, FILESIZE int-exp2]

[, EXTENDSIZE int-exp3]

[, WINDOWSIZE int-exp4]

[, USEROPEN sub-nam]

[, TEMPORARY]
```

(continued on next page)

```
[, CONTIGUOUS]
[, DEFAULTNAME str-exp2]
[,MAP map-nam]
```

Additional attributes for SEQUENTIAL files are:

```
[, BLOCKSIZE int-exp7]
[, NOREWIND]
[, NOSPAN]
```

Additional attributes for RELATIVE files are:

```
[, BUCKETSIZE int-exp8]
[, BUFFER int-exp9]
```

Additional attributes for INDEXED files are:

```
[, CONNECT chnl-exp2]
[, BUCKETSIZE int-exp10]
[, BUFFER int-exp11]
[, PRIMARY [KEY] vbl1 [DUPLICATES]]
[, ALTERNATE [KEY] vbl2 [DUPLICATES] [CHANGES] ]
```

Examples

```
10 OPEN "INPUT.DAT" FOR INPUT AS FILE #4%,  &
       ORGANIZATION SEQUENTIAL FIXED,  &
       ALLOW MODIFY, ACCESS MODIFY,
```

or

```
20 OPEN "TEMP.OUT" FOR OUTPUT AS FILE #3%
```

PRINT

The PRINT statement transfers program data to a terminal or a terminal-format file.

Format

```
PRINT [#chnl-exp] [exp [ { , } exp]. . .
                       { ; }
```

Example

```
100 PRINT "THE ANSWER IS "ICZ
```

or

```
200 PRINT EMP.NUM, EMP.NAME$, EMP.AGEZ
```

PRINT USING

The PRINT USING statement generates output formatted according to a format string. PRINT USING format strings specify either numeric or string formats.

Format

```
PRINT [# chnl-exp,] USING str-exp [,exp {;} ]...
```

Example

```
500 PRINT USING "$$*****.**-", 348832.433, -888.3, .67
```

PUT

The PUT statement transfers a record from record buffer to a file. Format 1 writes the next logical record to SEQUENTIAL, RELATIVE, or INDEXED files. Format 2 writes a specified record to a RELATIVE or VIRTUAL file.

Format 1

```
PUT #chnl-exp [, COUNT int-exp]
```

Example 1

```
100 PUT #3Z
```

Format 2

```
PUT #chnl-exp, RECORD int-exp [, COUNT int-exp]
```

Example 2

```
100 PUT #5Z, RECORD 133, COUNT 16%
```

RANDOMIZE

The RANDOMIZE statement gives the random number function, RND, a new starting point.

Format

```
{RANDOMIZE}
{RANDOM   }
```

Example

```
45 RANDOMIZE
```

READ

The READ statement assigns values from a DATA statement to variables.

Format

```
READ vbl [,vbl]. . .
```

Example

```
10 READ A, BZ, C$
20 DATA 32.5, 8, ENDDATA
```

REM

The REM statement allows you to write comments in your source program.

Format

```
REM [comment]
```

Example

```
500 REM THIS IS A COMMENT
```

RESTORE

The RESTORE statement resets the DATA pointer to the beginning of the DATA sequence, or sets the record pointer to the first record in a file.

Format

```
{RESTORE} [#chnl-exp [, KEY #int-exp]]
{RESET  }
```

Example

```
400 RESTORE #7%, KEY #4%
```

RESUME

The RESUME statement marks the end of an error-handling routine, and returns program control to a specified line number.

Format

```
RESUME [lin-num]
```

Examples

```
990 RESUME 300
```

or

```
990 RESUME
```

RETURN

The RETURN statement transfers control to the statement immediately after the most recently executed GOSUB or ON GOSUB statement.

Format

```
RETURN
```

Example

```
800 RETURN
```

RSET

The RSET statement assigns right-justified data to a string variable. It does not change the string variable's length.

Format

```
RSET str-vbl [,str-vbl. . .] = str-exp
```

Example

```
100 RSET ZZ$ = "LMNOP"
```

SCRATCH

The SCRATCH statement truncates a sequential file at the current record pointer.

Format

```
SCRATCH #chnl-exp
```

Example

```
600 SCRATCH #41
```

SLEEP

The SLEEP statement suspends program execution for a specified number of seconds.

Format

```
SLEEP int-exp
```

Example

```
60 SLEEP 120%
```

STOP

The STOP statement halts program execution.

Format

```
STOP
```

Example

```
95 STOP
```

SUB

The SUB statement marks the beginning of a BASIC subprogram and specifies its parameters by number and data type. The SUB statement accepts up to 32 arguments.

Format

```
SUB data-type sub-nam ([data-type] {param} [. . .]
                                   {,array-param [BY REF]} [. . .])
```

Example

```
10 SUB SUB3 (A, B, C$)
      .
      .
      .
90 SUBEND
```

SUBEND

The SUBEND statement marks the end of a BASIC subprogram and returns control to the calling program.

Format

```
SUBEND
```

Example

```
250 SUB HTFL
      .
      .
      .
32767 SUBEND
```

SUBEXIT

The SUBEXIT statement returns control to the calling program.

Format

```
SUBEXIT
```

Example

```
50 SUBEXIT
```

UNLESS

The keyword UNLESS follows a statement and is followed by a conditional expression. BASIC executes the statement only if the conditional expression is false.

Format

```
statement UNLESS cond-exp
```

Example

```
100 PRINT "A% DOES NOT EQUAL 3" UNLESS A% = 3%
```

UNLOCK

The UNLOCK statement unlocks the record or bucket locked by the latest FIND or GET.

Format

```
UNLOCK #chnl-exp
```

Example

```
90 UNLOCK #10%
```

UNTIL

The keyword UNTIL: (1) follows a statement and is followed by a conditional expression, or (2) marks the beginning of an UNTIL loop. If UNTIL follows a statement, BASIC executes the statement repeatedly until the conditional expression is true.

Format 1

```
UNTIL cond-exp
```

Example 1

```
10 UNTIL Z% * 100%
.
.
.
100 NEXT
```

Format 2

```
stmnt UNTIL cond-exp
```

Example 2

```
100 AZ = AZ + 1Z UNTIL AZ = 200Z
```

UPDATE

The UPDATE statement replaces a record in a file with the record in the record buffer.

Format

```
UPDATE #chnl-exp [, COUNT int-exp]
```

Example

```
100 UPDATE #4Z, COUNT 32Z
```

WAIT

The WAIT statement specifies the number of seconds the program waits for terminal input. The maximum WAIT time is 255 seconds.

Format

```
WAIT int-exp
```

Example

```
50 WAIT 60Z
60 INPUT "YOU HAVE SIXTY SECONDS TO TYPE YOUR NAME", NAME$
```

WHILE

The keyword WHILE: (1) follows a statement and is followed by a conditional expression, or (2) marks the beginning of an WHILE loop. If WHILE follows a statement, BASIC executes the statement repeatedly as long as the conditional expression is true.

Format 1

```
WHILE cond-exp
```

Example 1

```
10 WHILE X<100
20       X = X + SQR(X)
30 NEXT
```

Format 2

```
stmnt WHILE { log-exp }
            { rel-exp }
```

Example 2

```
100 X% = X% + 1% WHILE X% > 100%
```

A.4 Functions

This section describes the numeric and string functions available in BASIC.

ABS

The ABS function returns the absolute value of a specified numeric expression.

Format

```
num-vbl = ABS(num-exp)
```

Example

```
400 A = ABS(-100 * G)
```

ASCII

The ASCII function returns the decimal ASCII value of a string's first character.

Format

```
int-vbl = ASCII(str-exp)
```

Example

```
500 ASC.VAL% = ASCII(EMP.NAM$)
```

ATN

The ATN function returns the angle, in radians, of a specified tangent.

Format

```
num-vbl = ATN(num-exp)
```

Example

```
150 ANGLE = ATN(1)
```

CCPOS

The CCPOS function returns the current character or cursor position on a specified channel.

Format

```
int-vbl = CCPOS(chnl-exp)
```

Example

```
100 CHNLO% = CCPOS (0%)
```

CHR$

The CHR$ function returns a one-character string with the specified ASCII value.

Format

```
str-vbl = CHR$(int-exp)
```

Example

```
220 A$ = CHR$(65%)
```

COMP%

The COMP% function compares two numeric strings. If the first numeric string is less than the second, COMP% returns a minus one. If the strings are equal, COMP% returns a zero. If the first numeric string is greater than the second, COMP% returns a one.

Format

```
int-vbl = COMP%(num-str1, num-str2)
```

Example

```
400 NUM_STRING$ = "-.25E-12"  &
    \ OLD_NUM_STRING$ = "3"
450 ALPHA% = COMP%(NUM_STRING$, OLD_NUM_STRING$)
```

COS

The COS function returns the cosine of an angle you specify in radians.

Format

```
num-vbl = COS(num-exp)
```

Example

```
900 COSINE.ALPHA = COS(PI/2)
```

CTRLC

The CTRLC function enables CTRL/C trapping.

Format

```
int-vbl = CTRLC
```

Example

```
2000 YZ=CTRLC
```

DATE$

The DATE$ function returns the current date. DATE$ accepts only zero as an argument.

Format

```
int-vbl = DATE(0%)
```

Example

```
500 PRINT DATE(0%)
```

DET

The DET function returns the determinant of the last matrix inverted with the MAT INV function.

Format

```
num-vbl = DET
```

Example

```
100 DETERMINANT = DET
```

DIF$

DIF$ returns a string whose value is the difference between two numeric strings.

Format

```
str-vbl = DIF$(str-exp1, str-exp2)
```

Example

```
500 RESULT$ = DIF$("6778", "-455")
```

ECHO

The ECHO function causes characters typed at a terminal to be echoed on the terminal.

Format

```
int-vbl = ECHO(chnl-exp)
```

Example

```
100 Y% = ECHO(0%)
```

EDIT$

The EDIT$ function performs one or more editing operations, depending on the value of its integer argument.

Format

```
str-vbl = EDIT$(str-exp, int-exp)
```

Example

```
100 NEW.STRING$ = EDIT$(OLD.STRING$, 48%)
```

ERL

The ERL function returns the line number executing when the last error occurred.

Format

```
int-vbl = ERL
```

Example

```
300 IF ERL = 20% THEN CLOSE #2%
```

ERN$

The ERN$ function returns the name of the program or subprogram executing when the last error occurred.

Format

```
str-vbl = ERN$
```

Example

```
2000 IF ERR = 11% THEN RESUME 1000
```

ERR

The ERR function returns the number of the last error that occurred.

Format

```
int-vbl = ERR
```

Example

```
2000 ERROR.NUMBER% = ERR
```

ERT$

The ERT$ function returns explanatory text associated with a specified error number.

Format

```
str-vbl = ERT$(int-exp)
```

Example

```
2020 ERROR.TEXT$ = ERT$(11%)
```

65

EXP

The EXP function returns the value of the mathematical constant, "e," raised to a specified power.

Format

```
num-vbl = EXP(num-exp)
```

Example

```
100 A = EXP(4.6)
```

FIX

The FIX function truncates the value of a real number at the decimal point and returns the integer portion.

Format

```
num-vbl = FIX(num-exp)
```

Example

```
200 VALUE = FIX(-3.333)
```

FORMAT$

The FORMAT$ function converts a numeric value to a formatted string representation. The output string is formatted according to a string you provide.

Format

```
str-vbl = FORMAT$(num-exp, str-exp)
```

Example

```
440 Z$ = FORMAT$(A5/32, "**.**")
```

INSTR

The INSTR function searches for a substring within a string. It returns the substring's starting character position.

Format

```
int-vbl = INSTR(int-exp, str-exp1, str-exp2)
```

Example

```
300 YZ = INSTR(1%, ALPHA$, "JKLMN")
```

INT

The INT function returns a real number equal to the largest whole number that is less than or equal to a specified number.

Format

```
num-vbl = INT(num-exp)
```

Example

```
650 RESULT = INT(6.667)
```

LEFT$

The LEFT$ function extracts a substring from the left side of a string, leaving the string unchanged.

Format

```
str-vbl = LEFT[$](str-exp, int-exp)
```

Example

```
410 SUB.STRING$ = LEFT$(ALPHA$, 5%)
```

LEN

The LEN function returns the length of a specified string.

Format

```
int-vbl = LEN(str-exp)
```

Example

```
200 LENGTH% = LEN(ALPHA$)
```

LOC

The LOC function returns a longword integer specifying the absolute address of a variable or the value of an external symbol.

Format

```
lnt-vbl = LOC(symb)
```

Example

```
200 A% = LOC(B%)
```

LOG

The LOG function returns the natural logarithm of a specified number.

Format

```
num-vbl = LOG(num-exp)
```

Example

```
10 EXPONENT = LOG(100.35)
```

LOG10

The LOG10 function returns a number's common logarithm.

Format

```
num-vbl = LOG10(num-exp)
```

Example

```
600 EXP.BASE.10 = LOG10(250)
```

MAR

The MAR function returns the margin width of a specified channel.

Format

```
int-vbl = MAR(chnl-exp)
```

Example

```
200 WIDTHZ = MAR(0%)
```

MID$

The MID$ function extracts a substring from the middle of a specified string, leaving the string unchanged.

Format

```
str-vbl = MID[$](str-exp, int-exp1, int-exp2)
```

Example

```
220 NEW.STRING$ = MID$(OLD.STRING$, 5%, 8%)
```

NOECHO

The NOECHO function disables echoing on a specified channel.

Format

```
int-vbl = NOECHO(chnl-exp)
```

Example

```
500 Y% = NOECHO(0%)
```

NUM

The NUM function returns the row number of the last data element transferred into an array by a MAT statement.

Format

```
int-vbl = NUM
```

Example

```
10 ROW.COUNT% = NUM
```

NUM2

The NUM2 function returns the column number of the last data element transferred into an array by a MAT statement.

Format

```
int-vbl = NUM2
```

Example

```
COLUMN.COUNT% = NUM2
```

NUM$

The NUM$ function converts a numeric expression to the same string format as that of the PRINT statement.

Format

```
str-vbl = NUM$(num-exp)
```

Example

```
660 NUMBER$ = NUM$(34.5*5000/32.4)
```

NUM1$

The NUM1$ function changes a numeric expression to a string of numeric characters. NUM1$ does not return leading or trailing spaces, or E format.

Format

```
str-vbl = NUM1$(num-exp)
```

Example

```
750 NUMBER$ = NUM1$(PI/2)
```

PLACE$

The PLACE$ function changes the precision of a numeric string. BASIC rounds or truncates the numeric string according to the value of the integer argument.

Format

```
str-vbl = PLACE$(num-str, int-exp)
```

Example

```
500 NUMBER$ = PLACE$(OLD.NUMBER$, 10001%)
```

POS

The POS function searches for a substring within a string. It returns the substring's starting character position.

Format

```
int-vbl = POS(str-exp1, str-exp2, int-exp)
```

Example

```
400 YZ = POS(ALPHA$, "JKLMN", 12)
```

PROD$

The PROD$ function returns a numeric string that is the product of two numeric strings. The precision of the returned numeric string depends on the value of an integer argument.

Format

```
str-vbl = PROD$(num-str1, num-str2, int-exp)
```

Example

```
300 PRODUCT$ = PROD$("88793", Z$, 02)
```

QUO$

The QUO$ function returns a numeric string that is the quotient of two numeric strings. The precision of the returned numeric string depends on the value of an integer argument.

Format

```
str-vbl = QUO$(num-str1, num-str2, int-exp)
```

Example

```
200 QUOTIENT$ = QUO$("453.221", "30", 10000%)
```

RCTRLC

The RCTRLC function disables CTRL/C trapping.

Format

```
int-vbl = RCTRLC
```

Example

```
200 YZ = RCTRLC
```

RCTRLO

The RCTRLO function cancels the effect of a CTRL/O typed on a specified channel.

Format

```
int-vbl = RCTRLO (chnl-exp)
```

Example

```
250 YZ = RCTRLO(0%)
```

RECOUNT

The RECOUNT function returns the number of characters read by the last input operation.

Format

```
int-vbl = RECOUNT
```

Example

```
200 CHARACTER.COUNT% = RECOUNT
```

RIGHT$

The RIGHT$ function extracts a substring from the right side of a string, leaving the string unchanged.

Format

```
str-vbl = RIGHT[$](str-exp, int-exp)
```

Example

```
600 NEW.STRING$ = RIGHT$(ALPHA$, 21%)
```

RND

The RND function returns a random number greater than or equal to zero, and less than one.

Format

```
num-vbl = RND
```

Example

```
990 RNUM = RND
```

SEG$

The SEG$ function extracts a substring from a string, leaving the string unchanged.

Format

```
str-vbl = SEG$(str-exp, int-exp1, int-exp2)
```

Example

```
300 CENTER$ = SEG$(ALPHA$, 5%, 20%)
```

SGN

The SGN function operates on a numeric expression. It returns a one if the expression is positive, a minus one if the expression is negative, and zero if the expression is zero.

Format

```
int-vbl = SGN(num-exp)
```

Example

```
750 SIGN% = SGN(-45*35/6-3000)
```

SIN

The SIN function returns the sine of an angle specified in radians.

Format

```
num-vbl = SIN(num-exp)
```

Example

```
10 S1.ANGLE = SIN(PI/2)
```

SPACE$

The SPACE$ function returns a string containing a specified number of spaces.

Format

```
str-vbl = SPACE$(int-exp)
```

Example

```
880 FILLER$ = SPACE$(32%)
```

SQR

The SQR function returns a number's square root.

Format

```
num-vbl = SQR(num-exp)
```

Example

```
425 ROOT = SQR(35*37)
```

STATUS

The STATUS function returns an integer value containing information about the last opened channel.

Format

```
int-vbl = STATUS
```

Example

```
150 YX = STATUS
```

STRING$

The STRING$ function returns a string containing a specified number of identical characters.

Format

```
str-vbl = STRING$(int-exp1, int-exp2)
```

Example

```
340 F23$ = STRING$(10Z, 65X)
```

STR$

The STR$ function changes an number to a numeric string.

Format

```
str-vbl = STR$(num-exp)
```

Example

```
800 Z$ = STR$(G5)
```

SUM$

SUM$ is a string arithmetic function. It returns a string whose value is the sum of two numeric strings.

Format

```
str-vbl = SUM$(str-exp1, str-exp2)
```

Example

```
500 SIGMA$ = SUM$("234.444", A$)
```

SWAP%

The SWAP% function transposes an integer's two low-order bytes.

Format

```
int-vbl = SWAP%(int-exp)
```

Example

```
500 S_25% = SWAP%(3%)
```

TAB

When used with the PRINT statement, the TAB function moves the cursor or print mechanism rightwards to a specified column.

Format

```
PRINT TAB(int-exp)
```

Example

```
200 PRINT A$, TAB(15%), B$, TAB(30%), C$
```

TAN

The TAN function returns the tangent of an angle you specify in radians.

Format

```
num-vbl = TAN(num-exp)
```

Example

```
550 X = TAN(2*PI)
```

TIME

The TIME function returns the time of day (in seconds) as a real number. If the argument is zero, TIME returns the number of seconds elapsed since midnight. If the argument is one, TIME returns the current job's CPU time in tenths of a second. If the argument is two, TIME returns the current job's connect time in minutes. If the argument is three or four, TIME returns zero. All other arguments to TIME are undefined and cause BASIC to signal "Not implemented" (ERR=250).

Format

```
num-vbl = TIME(num-exp)
```

Example

```
150 PRINT TIME(0)
```

TIME$

The TIME$ function returns a string displaying the time of day in the form HH:MM AM or HH:MM PM. If the argument is zero, TIME$ returns the current time of day. If the argument is nonzero, TIME$ returns the time of day at that number of seconds before midnight.

Format

```
str-vbl = TIME$(int-exp)
```

Example

```
200 CURRENT_TIME$ = TIME$(0%)
```

TRM$

The TRM$ function removes trailing blanks and tabs from a specified string.

Format

```
str-vbl = TRM$(str-exp)
```

Example

```
600 NEW.STRING$ = TRM$(OLD.STRING$)
```

VAL

The VAL function returns a numeric string's real value.

Format

```
num-vbl = VAL(num-str)
```

Example

```
100 REAL = VAL("990.32")
```

VAL%

The VAL% function returns a numeric string's integer value.

Format

```
int-vbl = VAL%(int-str)
```

Example

```
100 A% = VAL%("999")
```

XLATE

The XLATE function translates one string to another by referencing a table you supply. For Example, the XLATE function translates from EBCDIC to ASCII.

Format

```
str-vbl = XLATE(str-exp1, str-exp2)
```

Example

```
100 OUTPUT$ = XLATE(INPUT$, TABLE$)
```

Table A–1: Arithmetic Operators

Operator	Use	Meaning
^ or **	5^2 or 5**2	exponentiation
*	A * B	multiplication
/	A /B	division
+	A + B	addition, unary plus, string concatenation
-	A - B	subtraction, unary minus

Table A–2: Logical Operators

Operator	Use	Meaning
NOT	NOT A	logical negative of A
AND	A AND B	logical product of A and B
OR	A OR B	logical sum of A and B
XOR	A XOR B	logical exclusive OR of A and B
EQV	A EQV B	logical equivalence between A and B
IMP	A IMP B	logical implication of A and B

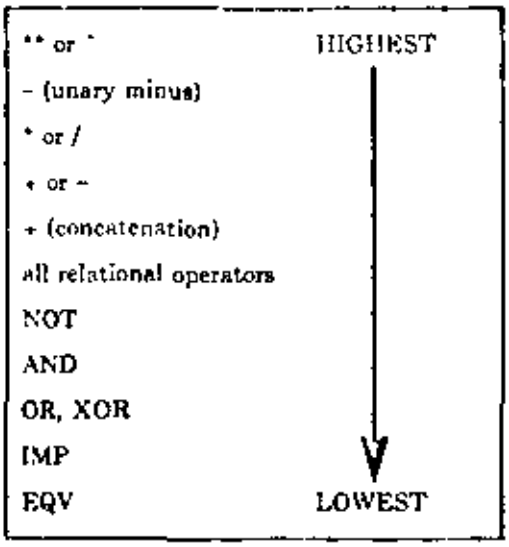
Table A-3: Relational Operators

Operator	Use	Meaning
=	A = B	A is equal to B
<	A < B	A is less than B
>	A > B	A is greater than B
<= or =<	A <= B	A is less than or equal to B
>= or =>	A >= B	A is greater than or equal to B
<> or ><	A <> B	A is not equal to B
==	A==B	A is approximately equal to B

Note that A is approximately equal to B (A==B) if the difference between A and B is less than 10^(-6). If A$ and B$ are strings, the relation (==) is true if the contents of A$ and B$ are the same in length and composition.

1. Operators on the same line in the table have left-to-right precedence.
2. The operators + and * are evaluated in algebraically correct order.
3. BASIC evaluates A^B^C as (A^B)^C.

Table A-4: Operator Precedence



BASIC evaluates expressions enclosed in parentheses first, even when the operator in parentheses has a lower precedence than that outside the parentheses. BASIC evaluates the expression:

```
A = 15^2+12^2-(35*8)
```

in five ordered steps:

1. 35*8=280
   Multiplication
2. 15^2 = 225
   Exponentiation (left-most expression)
3. 12^2 = 144
   Exponentiation
4. 225+144= 369
   Addition
5. 369-280 = 89
   Subtraction

Chapter 1 will give you an introduction to the FORTRAN language. In Chapter 2, you will see how to run a FORTRAN program on the computer.

ALGORITHMS

People are often imprecise when giving instructions to each other. A simple instruction such as "Go to the store for a loaf of bread," requires hundreds of decisions to carry out. Should you go out the front door or the back? Should you turn left or right? Which way is the store? Where is the bread? Do you want whole-wheat, sourdough, or caraway rye? People, having intelligence and reasoning ability, can figure out the real meaning of general, ambiguous instructions. Computers, on the other hand, have no common sense. When you write a procedure for a computer to carry out, every instruction must be explicit.

Suppose you want to solve a math problem using a calculator. You do not have a calculator, but you have a friend who does, so you call her on the phone to ask for help. She is not at home, but her young brother, who knows very little math, offers to work the calculator if you will tell him exactly what to do. Now you must specify how to solve your problem using instructions that are so precise and unambiguous that he cannot possibly misinterpret them. You might say, "Enter the number 56.2, press the plus key, enter the number 475.3, press the plus key, enter the number 11.63, press the equals key, and read me the number in lights at the top." This is an *algorithm*, expressed in English.

An **algorithm*** is a step-by-step procedure for solving a problem. A correct algorithm must meet three conditions:

1. Each step in the algorithm must be an instruction that can be carried out.
2. The order of the steps must be precisely determined.
3. The algorithm must eventually terminate.

An algorithm does not necessarily have to be written for a computer. You could carry out the instructions with a paper and pencil, for example. A cookbook might give an "algorithm" for baking chocolate chip cookies. The instruction "Bake until done." might be meaningful for a human cook, so it satisfies the first condition for an algorithm.

It would be convenient if you could just tell a computer "Solve the following problem . . ." and make the machine obey. A computer, however, has an instruction set consisting of only a hundred or so basic commands that it can carry out electronically. These commands are similar to the commands you can give to a calculator by pressing the keys. For example, you can tell a computer to add two numbers. The first condition for a correct algorithm means that when writing an algorithm for a computer, each step must be something that a computer can carry out by executing these basic instructions.

In carrying out the instructions in an algorithm, the machine performs one instruction at a time. The second condition for a correct algorithm means that the algorithm must precisely specify the order in which the instructions are performed.

The third condition means that an algorithm must not go on forever. The following procedure, then, is *not* an algorithm.

Procedure to Count
Step 1 Let N equal zero.
Step 2 Add 1 to N.
Step 3 Go to Step 2.

If you tried to have a computer carry out this procedure, the machine would, in theory, run forever. The following procedure, on the other hand, *is* an algorithm, because it will eventually terminate.

Algorithm to Count to One Million
Step 1 Let N equal zero.
Step 2 Add 1 to N.
Step 3 If N is less than 1,000,000, then go to Step 2; otherwise, halt.

TOP-DOWN DESIGN

When working on an algorithm for a simple problem, a solution may suddenly occur to you after just thinking about the problem for a while. Practical computer applications, however, are seldom so simple. Professional programmers commonly write programs consisting of thousands of computer instructions. Designing such a program can be as complicated as designing a machine with thousands of parts. In order for it to work, all the pieces must fit together in an organized framework.

Top-down design is an approach to the problem of designing an algorithm. It is a method you can use to organize your work and also to organize your algorithm.

In some ways, designing an algorithm is similar to writing an essay. When writing an essay, you begin with a general idea of what you want to say. You then proceed to organize your thoughts and choose words to convey your meaning to the reader. When designing an algorithm, you begin with a general idea of what you want it to do. You must then organize your ideas and choose the right sequence of instructions to the computer that make it carry out the desired actions. But in programming, as in writing, it is often difficult to know where to begin.

A good way to begin writing an essay is to make an outline. First, you set forth the main topics to be covered, as in the following example:

Gettysburg Speech
I. Conception of the nation
II. The current civil war
III. Our purpose here today
IV. Our resolve for the future

*The word algorithm comes from the name of the Persian mathematician al-Khowarizmi (c. 825).

This bare outline provides a framework for the essay, into which all the paragraphs and sentences will fit. The overall structure is determined, and what remains is to elaborate the topics in more detail. You can do that by adding subheadings under each topic, as in the following example:

Gettysburg Speech

I. Conception of the nation
   - A. The nation was founded 87 years ago.
   - B. It was conceived in liberty.
   - C. It was dedicated to the proposition that all men (persons?) are created equal.

II. The current civil war
   - A. We are now engaged in civil war.
   - B. The war tests the ability of this nation, as conceived by its founders, to endure.

III. Our purpose here today
   - A. We are meeting on a battlefield of the civil war.
   - B. We dedicate a portion of this field as a cemetery.
   - C. The soldiers who fought here have consecrated this ground with their brave struggle.

IV. Our resolve for the future
   - A. We must dedicate ourselves to the unfinished work before us.
   - B. We must devote ourselves to the cause for which the dead have fought.
   - C. We must preserve the ideals of freedom in which the nation was conceived.

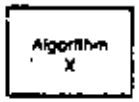
Of course, even with the best of outlines to work from, one cannot hope always to emulate Lincoln's death'em prose, but good writing is always well organized, and making an outline is a very useful way to begin.

In programming, as in writing, good organization is vital. Before beginning to fill in the details, it is best to have a clear overall plan. The method of top-down design is like making an outline of an algorithm. The first step is to formulate a general plan—like writing the main topics in an outline. Next, you fill in more detail, like making subheads in an outline, to specify how to carry out each major step. By successively refining this plan, adding more detail at each stage in the development, you arrive at your ultimate goal: a precise algorithm that can be expressed in terms of the basic types of instructions that a computer can follow.

As an illustration, suppose you want to write an algorithm (we'll call it Algorithm X), to solve some homework problem for your algebra class. As you begin, you have a rough idea of what the algorithm should do, but little idea of the specific instructions. At this point, you conceive of Algorithm X as a single entity (Figure 1.1).

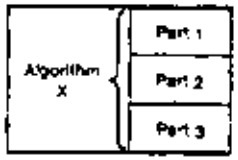
As you think about the problem some more, you realize that there are really three parts to the solution, and you say to yourself, "If I could do Part 1, and then Part 2, and Part 3, that would solve the problem." This is your top-level design, as shown in Figure 1.2.

FIGURE 1.1



As you begin to write an algorithm you conceive of it as a single entity.

FIGURE 1.2



In the method of top-down design, you first specify the general organization of the algorithm.

You may not have a detailed algorithm yet, but you have made some progress: instead of one big problem, you have three smaller ones to solve. Continuing your analysis, you might find that Part 1 can be broken down into two subproblems—Part 1A and Part 1B—and that Parts 2 and 3 can be similarly subdivided. This is your second-level design, as shown in Figure 1.3. If the algorithm is complicated, you may need to carry this process to yet another level of refinement. Eventually, you arrive at a detailed plan that you can write in computer language.

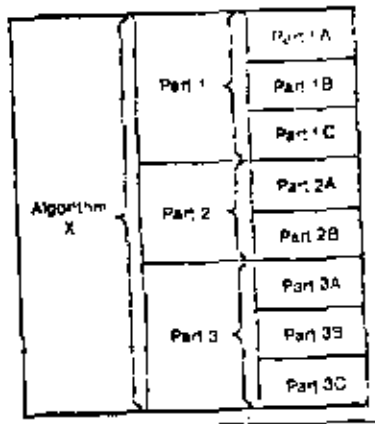
In summary, the basic principle of top-down design is this:

> Concentrate first on the overall design of the algorithm. Write it as a sequence of general steps to be carried out. Then, using the same method, fill in the details for each step.

An important principle in top-down design is validation, which means analyzing your algorithm to make sure that it is correct. Even though your first draft of an algorithm may be an outline, it must be a valid solution to the problem you are trying to solve. The question you should ask yourself is this: If you could carry out each step in your proposed algorithm, would that really solve the problem? Thus, a second principle of top-down design is this:

> At each stage of the top-down design process, validate your algorithm by analyzing it for correctness. Do not expand on the details until you are sure that the overall plan is sound.

FIGURE 1.3



You refine the top-level design by adding more detail.

EXAMPLE 1.1 Averaging Numbers

Suppose you were given the task of finding the average of the numbers 5.2, 11.35, 4.0, 6.7, and 5.9. How would you do it? Using a calculator, you would probably enter the first number, add the second number to it, add the third number to the sum, add the fourth number to that sum, then add the fifth number. Mentally observing that there are five numbers, you would divide by 5 to find the average. What you would really be doing is carrying out the steps in a familiar algorithm for averaging numbers.

Now suppose you want to write a general algorithm that could be used to instruct a computer to find the average of a set of numbers and print out the result. To begin very simply, there are two main steps in the algorithm.

Algorithm for Averaging Numbers (Version 1)

I. Compute the average.
II. Print the average.

This is your top-level design.

To fill in more detail, you need to describe the procedure to compute the average as you would do it with a calculator. Here is a revised version.

Algorithm for Averaging Numbers (Version 2)

I. Compute the average.
  A. *Read* each number. As you go through the list, compute the sum of the numbers and count them.
  B. Compute the average, using the sum and the count from Step IA.
II. Print the average.

This is your second-level design.

Next you should validate the algorithm to check your work. You should ask yourself: Is it really possible to carry out Step IA? Is it really possible to carry out Step IB? Do these two steps together really compute the average required in Step II? Theoretically, you can use this type of reasoning to construct a mathematical proof that an algorithm is correct. For most practical purposes it is sufficient to go through an informal mental proof to convince yourself that you haven't made any mistakes.

In validating Step IB, there is one small "catch" to consider. To compute the average you must divide the sum of the numbers by the count. What would happen if the count were zero? Dividing a number by zero is mathematically undefined, and will cause problems if you try to do it on a computer. If you were averaging numbers with a calculator, the question would not arise, since you, as a human, have the common sense not to divide by zero. When writing an algorithm for a computer, however, it is wise to consider exceptional cases, like carrying out the algorithm with no input data. Such exceptional cases are called **boundary conditions**. With a little extra effort, you can design algorithms that work even for the boundary conditions. This practice will simplify the job of checking the algorithm on a computer, as you will soon appreciate when you begin running your own programs.

You can now refine the algorithm to fill in the details of the computation.

Algorithm for Averaging Numbers (Version 3)

I. Compute the average.
  A. Count the numbers and find their sum.
    1. Let SUM and COUNT be zero.
    2. For each number to be averaged, do the following.
      a. Read the number.
      b. Add its value to SUM.
      c. Add 1 to COUNT.
  B. Compute the average.
    1. *If* COUNT is not zero, *then*
      a. Let AVERAGE equal SUM divided by COUNT.
      Otherwise,
      b. Let AVERAGE equal zero.
II. Print the value of AVERAGE.

This is a complete algorithm which gives enough detail to tell you exactly how to carry out the computation. As before, you should validate this algorithm to convince yourself that Steps IA1 and IA2 really do count the numbers and find their sum and that Step IB really does compute the average (and that it computes an average of zero for the exceptional case when COUNT is zero).

This algorithm illustrates several common programming techniques. The words COUNT, SUM, and AVERAGE are variables which represent numbers used in the computation. In programming, as in algebra, you use variables to stand for numbers whose specific values are not known.

Step IA1 says to start with SUM and COUNT equal to zero. This practice is called **initializing the variables**—that is, giving them an initial value. This is somewhat like pressing the CLEAR key on a calculator before starting a new computation.

Step IA2 is an example of a **loop**, a procedure that is repeated over and over. Each execution of the instructions in this step is called an **iteration** of the loop (to *iterate* means to do again). The loop is controlled by a condition. In this case the condition is whether or not there

are any more numbers to be averaged. When the condition is false, after all the numbers have been read, the algorithm breaks out of the loop and proceeds to Step I8.

The variable COUNT is used to count the number of values read. Any variable used in this way is called a **counter**. The process of adding 1 to COUNT is called **incrementing** the counter. The algorithm increments the counter on each iteration of the loop.

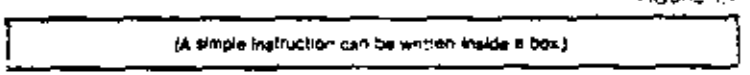
Step I81 in the algorithm is a **conditional instruction** (often called an **if-then-else instruction**) which says to carry out either Step I81a or I81b depending on the condition "COUNT is not equal to zero."

STRUCTURED FLOWCHARTS

A **flowchart** is an easy-to-read diagram of an algorithm. You write each step inside a box and assemble the boxes to show the order in which the instructions are to be carried out. The kind of flowcharts in this book are called **structured flowcharts**.* These flowcharts facilitate the technique of structured programming explained in Chapter 4.

You diagram a simple instruction, like initializing a variable, inside a rectangular box (Figure 1.4).

FIGURE 1.4



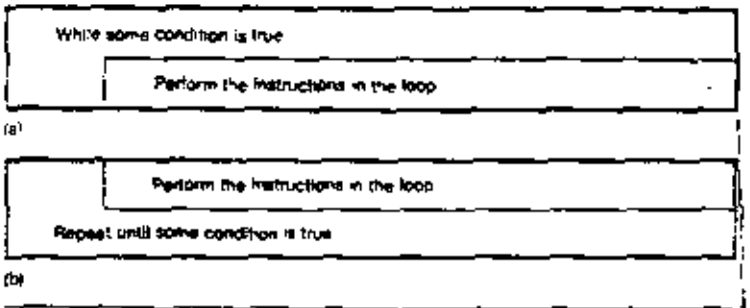
To indicate a loop in an algorithm, you use an L-shaped box that surrounds the instructions in the loop (Figure 1.5). The condition in the outer box is the condition that controls the loop. This indicates that the instructions in the loop are to be performed while the condition is true. You can test the condition either at the beginning of the loop [Figure 1.5(a)] or at the end [Figure 1.5(b)]. For example, Step IA2 of the algorithm for averaging numbers is a loop which you can diagram as in Figure 1.6.

To indicate a conditional instruction you use a triangular shape (Figure 1.7). If the condition in the triangle is true, the algorithm carries out the instructions underneath the word "then." If the condition is false, the algorithm carries out the instructions underneath the word "else." For example, Step I31 of the algorithm for averaging numbers can be diagrammed as in Figure 1.8.

Thus, you can represent the entire algorithm to average numbers by the structured flowchart in Figure 1.9.

*Also called Nassi-Schneiderman diagrams; they are based on a method invented by I. Nassi and B. Schneiderman [ACM SIGPLAN Notices, Vol. 8, No. 8, August 1973].

FIGURE 1.5
TWO WAYS OF SHOWING A LOOP



In (a) the condition is tested at the beginning of the loop, while in (b) the condition is tested at the end.

FIGURE 1.6

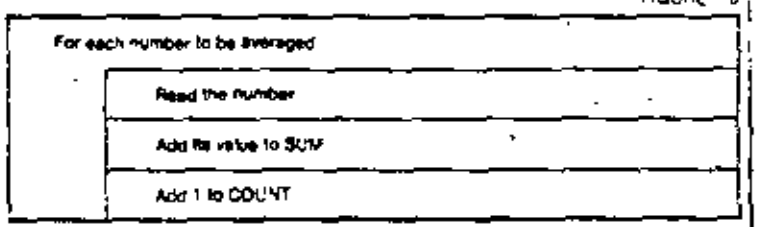


FIGURE 1.7

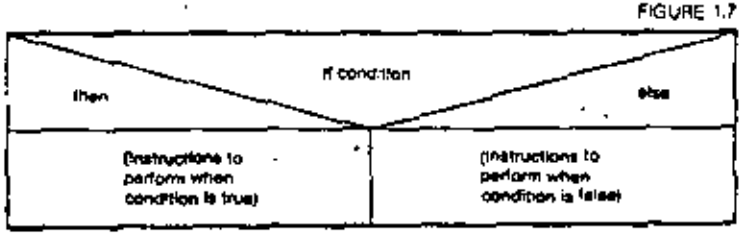


FIGURE 1.8

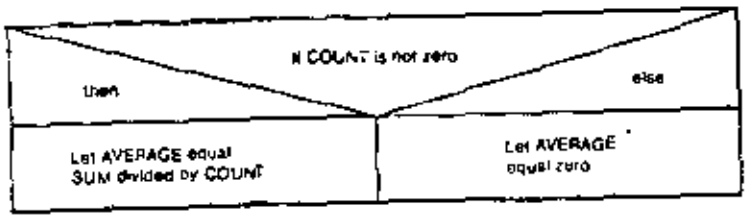
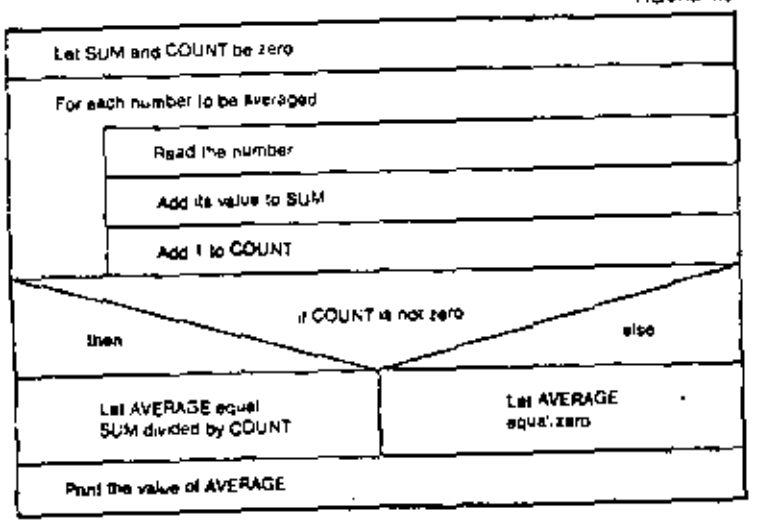


FIGURE 1.9



THE FORTRAN LANGUAGE

Inventing an algorithm is the first step in solving a problem with a computer. You can write the algorithm in outline form,* as on page 7, or you can diagram the algorithm with a structured flowchart. The principles of top-down design apply in either case.

The next step is to write a program. A program is an algorithm written in a computer language. In this book, you will learn the FORTRAN language.* Compared to ordinary English, FORTRAN is a simple language. The vocabulary consists of only a few dozen words (plus any variable names you make up yourself), and you can get by if you know fewer than a dozen types of sentences. The sentences in FORTRAN are called statements. A program consists of a series of statements, each one written on a separate line.

Though FORTRAN is simple, it is precise. If you write an English sentence that is grammatically incorrect, the chances are that your reader will understand you anyway. Not so in FORTRAN. Every statement has a certain required form. Leave out a comma where one is required in FORTRAN and the computer will fail to recognize the statement or else will misinterpret it.

Because FORTRAN is precise and unambiguous, the computer can read your program, translate it into commands in its instruction set, and carry out the steps in your algorithm. In fact, the name FORTRAN stands for FORmula TRANslation. What we now call "statements" were originally called "formulas." Thus, FORTRAN is a language for translating the statements in your program into instructions that the machine can carry out.

The following sections introduce some basic concepts of FORTRAN and some simple FORTRAN statements so that you can begin running programs right away. For the moment, we will skip over many details. Just as you first learned English by listening to people speak, you can learn something about FORTRAN by seeing some complete programs. We will cover the exact rules in later chapters.

WHAT COMPUTERS DO

To understand the instructions that make up a FORTRAN program, you need a basic knowledge of what a computer is and what it does.

Figure 1.10 is a simplified block diagram of a typical computer system. It is called a *system* since it consists of several components. The central processor is the actual computer. This is the machine that carries out the instructions in your program. The memory unit is where instructions and data are stored. The other components are peripheral devices used for input and output.

The computer, although very fast, is really a very simple machine. There are only four basic types of instructions that it can carry out:

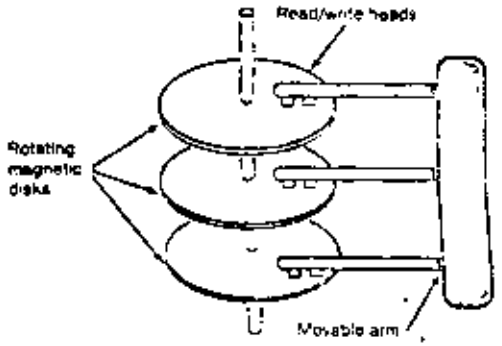
1. storing and recalling from memory
2. computation
3. input and output
4. program control

All programs, no matter how complex, are made up of instructions of these types, along with declarative statements that help the computer interpret the instructions.

*Another name for this "outline form" is pseudocode—so called because it resembles a real "code" (that is, a FORTRAN program) but is not a complete program.

*To be specific, you will learn FORTRAN 77 (see the Preface).

FIGURE 1.17
SCHEMATIC DIAGRAM OF A MAGNETIC DISK UNIT



The movable read/write heads are electromagnetics that selectively magnitize areas of the rotating disks

The FORTRAN instruction for input is the READ statement. An example is

```
READ *, VALUE
```

This instruction says to read a number and store it in the variable VALUE. Every time you read a number you must store it in some variable.

The FORTRAN instruction for output is the PRINT statement. An example is

```
PRINT *, TOTAL
```

which means to print the value of the variable TOTAL. The statement

```
PRINT *, 'THE SUM IS', SUM
```

means to print the words 'THE SUM IS', followed by the value of the variable SUM. If SUM is equal to 100, this statement will print the output line:

```
THE SUM IS 100.0
```

Tables 1.2 and 1.3 give some more examples.

TABLE 1.2
THE READ STATEMENT

Form

READ *, *list*

where *list* is a list of variable names. If there is more than one variable, separate them by commas

Meaning

Read values from the input unit (cards or terminal), and store the values in the variables listed.

Examples

```
READ *, A
READ *, CONVER, FACTOR, NUMBER
READ *, X, NUMBER
```

TABLE 1.3
THE PRINT STATEMENT

Form

PRINT *, *list*

where *list* is a list of items to be printed. If there is more than one item in the list, separate them by commas. Each item can be a variable, a title enclosed in single quote marks, or a mathematical expression.

Meaning

Write the values in the list on the standard output unit (printer or terminal).

Examples

```
PRINT *, 'THE ANSWER IS ', B
PRINT *, PCT, 'PERCENT'
PRINT *, 'ALL NUMBERS ARE IN CENTIMETERS'
PRINT *, A, B, C, D
PRINT *, A, B, ' SUM = ', A + B
```

Control Instructions

The statements in a FORTRAN program are normally carried out in the order in which they are written. You can modify that order with control instructions.

To represent a decision, for example, you can use an IF-THEN-ELSE construct of the form

```
IF (condition) THEN
    statement 1
ELSE
    statement 2
END IF
```

In this example, *statement 1* and *statement 2* represent any FORTRAN statements, and *condition* represents some simple condition that a computer can test electronically to determine whether it is true or false.

For example, a computer can test a variable A and a variable B to see if their values are equal. You would write this condition as

```
A .EQ. B
```

The symbol ".EQ." is the FORTRAN symbol for "equal to." Similarly, ".NE." means "not equal to."

In the algorithm for averaging numbers, one of the steps said, "If COUNT is not zero, then let AVERAGE equal SUM divided by COUNT; otherwise, let AVERAGE equal zero." The corresponding FORTRAN statements are:

```
IF (COUNT .NE. 0) THEN
    AVERAG = SUM/COUNT
ELSE
    AVERAG = 0
END IF
```

(See Table 1.4.) Note that we dropped the E from AVERAGE since FORTRAN variables can be at most six characters long.

TABLE 1.4
THE IF-THEN-ELSE STATEVENTS

Form

```
IF (condition) THEN
    statement 1
ELSE
    statement 2
END IF
```

Meaning

If the condition is true, execute *statement 1*; otherwise, execute *statement 2*.

Example

```
IF (COUNT .NE. 0) THEN
    AVERAG = SUM/COUNT
ELSE
    AVERAG = 0
END IF
```


Another kind of control statement is the GO TO statement, which you can use for a loop. Here is an example:

```
10  READ *, X
    PRINT *, X
    GO TO 10
```

The number "10" here is a statement label. The first two statements read and print a number, using a variable X to store it. The GO TO statement says to return to statement 10, the READ statement. (See Table 1.5.)

TABLE 1.5
THE GO TO STATEMENT

Form

```
GO TO n
```

where *n* is a statement label

Meaning

Branch to the statement with statement label *n*.

Examples

```
GO TO 1000
GO TO 7
GO TO 99
```

In the algorithm for averaging numbers there is a loop that is performed for every input value. You can write this procedure in FORTRAN by using a GO TO statement in conjunction with a special form of the READ statement, as follows.

```
10  READ (*, *, END = 20) VALUE
    SUM = SUM + VALUE
    COUNT = COUNT + 1
    GO TO 10
20  IF (COUNT .NE. 0) THEN
```

This form of the READ statement is a combination input and control instruction. If there is more input data, the READ statement reads the next number, stores it in the variable VALUE, and proceeds to the following assignment statement. However, if there is no more input data, the READ statement causes the program to branch to statement 20, the IF statement. That is the meaning of the symbol "END = 20" in the READ.

The last line in any FORTRAN program is an END statement, which you write simply as

```
END
```

This instruction indicates the end of your program and tells the computer to

Putting It All Together

Using the statements just covered, you can write a complete program. The following program finds the average of a set of numbers. We have added a few small embellishments to the algorithm given earlier. First, we included a PRINT statement right after the READ statement to print each input value. This technique is called echo checking. This way, you get a listing of your data along with the output. Second, we included PRINT statements at the beginning to provide titles for the output. Finally, we included statements to print the sum and the count of the numbers as well as the average. Each of these three values will be labeled in the output. Here is the complete program.

```
      INTEGER COUNT
      REAL VALUE, SUM, AVERAG
      PRINT *, 'AVERAGE NUMBER PROGRAM'
      PRINT *, 'INPUT VALUES:'
      SUM = 0
      COUNT = 0
10    READ (*, *, END = 20) VALUE
      PRINT *, VALUE
      SUM = SUM + VALUE
      COUNT = COUNT + 1
      GO TO 10
20    IF (COUNT .NE. 0) THEN
         AVERAG = SUM/COUNT
      ELSE
         AVERAG = 0
      END IF
      PRINT *, 'THE SUM IS ', SUM
      PRINT *, 'THE COUNT IS ', COUNT
      PRINT *, 'THE AVERAGE IS ', AVERAG
      END
```

SUMMARY

An algorithm is a step-by-step procedure for solving a problem. Each step must be a meaningful instruction. The order in which the steps are performed must be precisely determined, and the algorithm must eventually terminate.

In the top-down method of designing an algorithm, you begin by describing the overall plan of your solution. You refine the top-level design by adding more detail. As you develop the algorithm, you continually validate it to make sure that it is correct.

A structured flowchart is a diagram of an algorithm. Instructions are written in boxes, and the boxes are arranged to indicate the order in which the instructions are carried out.

FORTRAN is a precise language that can be used to write an algorithm in a form that a computer can carry out. There are only a few different kinds of statements in FORTRAN, each of which has a certain prescribed form.

A typical computer system consists of a central processor, memory unit, and peripheral devices. The instructions that a computer can perform fall into four categories: storing and recalling from memory, computation, input and output, and program control.

The assignment statement in FORTRAN tells the machine to carry out computations and store the result in memory. Each variable in FORTRAN is a name for some memory location.

Several kinds of devices are used for input and output. The FORTRAN statements READ and PRINT tell the machine to perform input and output.

Control instructions specify the order in which instructions are carried out. The IF-THEN-ELSE statements indicate decisions in a FORTRAN program. The GO TO statement causes a branch to some statement and can be used to program a loop.

VOCABULARY

address
algorithm
assignment statement
boundary condition
card reader
computer
computer terminal
conditional instruction
counter
cathode-ray tube
echo checking
END statement
exit from a loop
flowchart
FORTRAN
GO TO statement
if-then-else
increment
initialize
input
instruction set
iterate
line printer
loop
magnetic disk
magnetic tape
memory
microfilm
output
peripheral device
program
program control instruction
pseudocode
statement
statement label
store and recall from memory
system
top-down design
validation
value of a variable
variable

Rule for FORTRAN Variable Names
A variable name must begin with a letter. It can consist of from one to six characters. Only letters (A – Z) and numbers (0 – 9) may be used in the name. Special symbols such as +, -, or / are not allowed.

Table 3.5 gives some examples of FORTRAN variable names.

TABLE 3.5
EXAMPLES OF VARIABLE NAMES

Some Correct Variable Names	
A	LETTER
I	ALPHA
MASS	WW777
TAX	XNN
X1234	HELLO

Some Incorrect Variable Names	
UKLMNOPQ	(Too many characters)
I+J	(Illegal symbol)
4H	(Does not start with a letter)
NAME.1	(Illegal symbol)
AMOUNTDUE	(Too many characters)

Like constants, each FORTRAN variable has a certain type. An integer variable stores integer values, and a real variable stores real values.

You can choose, for each variable in your program, whether to make it integer or real. If you make it an integer variable, it will be an integer variable throughout the program, and will contain only integer values. If you make it a real variable, it will be a real variable throughout the program, and will contain only real values. You can indicate your choice by using an INTEGER or a REAL statement.

Suppose that you want the variables COUNT, TICKS, and N to be integer variables. At the beginning of the program, you can write the FORTRAN statement

```
INTEGER COUNT, TICKS, N
```

Similarly, if you want the variables X, L, and BETSY to be real variables in a program, use the following statement at the beginning:

```
REAL X, L, BETSY
```

EXECUTABLE AND NONEXECUTABLE STATEMENTS

The INTEGER and REAL statements are classified as type statements: statements that tell the type (integer or real) of certain variables. Type statements are one kind of specification statement in FORTRAN. A specification statement gives information about the variables used in a program. You will learn about other specification statements later on.*

All specification statements are **nonexecutable**. Nonexecutable statements do not cause the machine to do anything when the program runs. They provide information about the program that is used when the program is compiled.

Assignment statements, READ statements, and PRINT statements, on the other hand, are examples of **executable statements**. Executable statements *do* cause something to happen when the program is run. Executable statements are your instructions to the computer. Nonexecutable statements are instructions to the FORTRAN compiler, giving supplemental information about how to carry out your instructions.†

We mentioned before that INTEGER and REAL statements go at the beginning of a program. To be more precise, the rule is this:

Rule for Location of Specification Statements
All the specification statements in a program must come before the first executable statement in the program.

IMPLICIT TYPE DECLARATION

The INTEGER and REAL statements let you *explicitly* declare the types of each variable in your program. There is another way to specify the type of a variable.

Suppose you decide to adopt the following convention in writing a certain program: All variables whose names begin with the letter "A" will be integer variables, and all variables whose names begin with the letter "B" will be real variables. Rather than listing each variable in an INTEGER or REAL statement, you can include the following IMPLICIT statement at the beginning of your program:

```
IMPLICIT INTEGER (A), REAL (B)
```

This specification statement defines variable types *implicitly*. The type of a variable is implied by its name. Variables named APPLE, or A, or ALVIN will be integer variables in the program, since their names begin with A. Variables named B12, or BILL, or BTU will be real variables, since their names begin with B.

An IMPLICIT statement consists of the word IMPLICIT followed by a type, which may be INTEGER, or REAL (or any of the other types you will study in this book), followed by a set of parentheses that tell which letters are associated with the type. You can use a single letter, as in

```
IMPLICIT INTEGER (X)
```

*The FORTRAN type statements are: INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, and CHARACTER. The other specification statements are: DIMENSION, EQUIVALENCE, COMMON, IMPLICIT, PARAMETER, EXTERNAL, INTRINSIC, and SAVE.

†In addition to the specification statements already listed, the following FORTRAN statements are nonexecutable: PROGRAM, FUNCTION, SUBROUTINE, ENTRY, BLOCK DATA, DATA, FORMAT, and statement function definition.

meaning that all variables beginning with letter X are integer variables. You can give a sequence of letters, separated by commas, as in

```
IMPLICIT INTEGER (A, B, C, D, E)
```

meaning that all variables beginning with letters A, B, C, D, or E are integer variables. Another way to write this is

```
IMPLICIT INTEGER (A - E)
```

In an IMPLICIT statement, the notation "A – E" means "A through E."

If you use an IMPLICIT statement, it must be the very first statement in your program. To be more precise, the rule is:

> *Rule for Position of Implicit Statements*
> All IMPLICIT statements must precede all other specification statements and all executable statements.

You can override (or confirm) the effect of an IMPLICIT statement by including an explicit type declaration. For example, the two statements

```
IMPLICIT REAL (A - Z)
INTEGER I, N
```

mean that all variables in the program will be real variables, *except* for I and N, which are integer variables.

DEFAULT TYPE DECLARATION

FORTRAN does not require that every variable be declared in a type statement. When you do not declare the type of a variable, the variable has a default type, either integer or real, depending on the variable's name.

> *Rule for Default Types*
> Unless otherwise specified (by a type statement or an IMPLICIT statement), if a variable name begins with the letter I, J, K, L, M, or N, it is an integer variable. If the name begins with a letter A through H, or O through Z, it is a real variable.

In other words, unless you specify otherwise, variable types are determined as though you began the program with the statement

```
IMPLICIT REAL (A - H), INTEGER (I - N), REAL (O - Z)
```


This convention comes from the standard practice in mathematics of using variables *i*, *j*, *k*, *l*, *m*, and *n* for subscripts or for variables that take only integer values.

STYLE MODULE—CHOOSING VARIABLE NAMES

It is usually a challenging mental exercise, when writing a program, to keep track of the variables you have used. Many bugs turn out to be due to a misunderstanding of the exact definition of a variable, or to using a variable in two slightly different ways in different parts of the program.

A good way to avert this problem is to choose variable names that mean something. If you need a variable to represent time, you could call it T, but a better choice is TIME. If a variable represents a street address, call it STREET, rather than A. Avoid variable names that are too similar. If your program uses variables for State and Federal Tax, you could call them TAX1 and TAX2; better names would be STATAX and FEDTAX. Do not use randomly chosen names like X9.

Another helpful practice is to write a definition of each variable and to include the dictionary as comments in your program. The program on page 74 gives an example. If you use top-down program design methods, making such a dictionary should be easy, since your top-level program outline will define the data being processed.

If your FORTRAN compiler can produce a cross-reference list of variable names, learn to use this feature. Glancing through such a list can help to quickly locate typographic errors. In a cross-reference list, some compilers will mark variables that have been used but not defined. For instance, if a variable X2 appears in only one statement such as

```
Y = X2
```

it is almost certainly an error. Perhaps the programmer intended to write

```
Y = X+2
```

As another example, if you see a variable named PRINT in your listing when you know that you have used no such variable (intentionally), you should suspect that some PRINT statement in the program was incorrectly typed.

STYLE MODULE—USING TYPE STATEMENTS

You have seen three ways to declare the type of a variable:

1. Use an explicit type statement (INTEGER or REAL).
2. Use the default type, which depends on the first letter of the name.
3. Use an IMPLICIT statement to create your own default types.

SYMBOLIC CONSTANTS

You have seen how to write FORTRAN constants like

77 (an integer constant)
3.5 (a real constant)

In each case, the constant is represented by a specific value. In science and mathematics it is common to represent a constant by a *symbol*. Some examples are:

π	The ratio of the circumference to the diameter of a circle
e	The base of natural logarithms
c	The speed of light

You can write the formula for the area of a circle as

$$A = 3.14159265r^2$$

or as

$$A = \pi r^2$$

In the first case, we used the *value* of the constant (rounded to an approximation). In the second case, we used the *symbol* π instead.

In FORTRAN, as in mathematics, it is often convenient to represent a constant by a symbol, rather than by a value. The PARAMETER statement gives you a way to do this. The following program illustrates its use.

```
      REAL RADIUS, PI
      PARAMETER (PI = 3.14159265)
10    READ (*, *, END = 20) RADIUS
      PRINT *, 'RADIUS = ', RADIUS
      PRINT *, 'DIAMETER = ', 2*RADIUS
      PRINT *, 'CIRCUMFERENCE = ', 2*PI*RADIUS
      PRINT *, 'AREA = ', PI*RADIUS**2
      GO TO 10
20    END
```

The PARAMETER statement here declares that PI is a symbolic constant with a value of 3.14159265. The last two PRINT statements use this value of PI in expressions for the circumference and area of a circle. The effect is exactly as though we had written the statements with the numerical value of PI.

The PARAMETER statement is a specification statement. It is nonexecutable, and must come before the first executable statement in a program.

You can write a PARAMETER statement in the form

PARAMETER (*constant* = *expression*)

where *constant* is a symbolic constant name, and *expression* is a constant expression.

A symbolic constant name has the same form as a FORTRAN variable name. It consists of one to six letters and digits, and must begin with a letter.

Like a variable, a symbolic constant has a certain type (such as integer or real). You specify the type of a symbolic constant the same way you specify the type of a variable: with a type statement, an IMPLICIT statement, or with a default implicit type. If you use a type statement to give the type of the symbolic constant, the type statement must come before the PARAMETER statement that defines the value of the symbolic constant. Thus, if you want SIZE to be a symbolic constant representing the integer value 100, you could use the statements

```
INTEGER SIZE
PARAMETER (SIZE = 100)
```

To the right of the equal sign in a PARAMETER statement, you write a constant expression, which is an expression that involves only constants.

You can define several symbolic constants with a single PARAMETER statement by separating the definitions with commas, as in

```
INTEGER ROWS, COLS, ZERO
PARAMETER (ROWS = 20, COLS = 100, ZERO = 0)
```

A constant expression in a PARAMETER declaration may involve symbolic constants that have been previously defined. For example:

```
INTEGER NCOL, UNSIZ, PAGSIZ
PARAMETER (NCOL = 80)
PARAMETER (UNSIZ = NCOL + 1, PAGSIZ = 50*UNSIZ)
```

Here is a complete program that uses PARAMETER statements.

```
* CONVERT YEARS TO SECONDS
      REAL SECMIN, SECHR, SECDAY, SECYR
      PARAMETER (SECMIN = 60.)
      PARAMETER (SECHR = 60.*SECMIN)
      PARAMETER (SECDAY = 24.*SECHR)
      PARAMETER (SECYR = 365.*SECDAY)
      READ *, YEARS
      PRINT *, YEARS, 'YEARS EQUALS ',
     $    SECYR*YEARS, ' SECONDS.'
      END
```

As this program illustrates, a constant expression need not be the same type as the symbolic constant being defined. The rules here are the same as for an assignment statement. Thus in the first PARAMETER statement in this example, the integer value 60 is converted to a real value to become the value of the symbolic constant SECMIN.

There are two restrictions on constant expressions: you cannot use exponentiation unless the exponent is an integer, and you cannot use function references (which we discuss in Chapter 7).

You may wonder, at this point, what the difference is between a symbolic constant, defined by a PARAMETER statement, and a variable defined by an assignment statement. In many respects, there is little difference. You can define a constant LIMIT with the specification statement

```
PARAMETER (LIMIT = 25 - 25)
```

or you can define a variable LIMIT with the executable statement

```
LIMIT = 25 - 25
```

One difference is that once the value of a symbolic constant is defined, you cannot redefine it, deliberately or accidentally. An even more important difference is that symbolic constants are permissible in certain cases where variables are not. For instance, you can use a constant expression to define the length of character variables, as you will see in Chapter 5.

STYLE MODULE—SYMBOLIC PROGRAMMING

It is good programming practice to use symbols, rather than numbers, for quantities that might change.

Suppose you are writing a program to update account balances for a Savings and Loan Company. Just as you finish, the president walks by your desk and says, "By the way, our interest rate on passbook savings is going up 1/4 percent next week. Will that cause you any problems in your program?" Ideally, you should be able to reply, "Not at all, I'll just change one line." You might have followed a convention in your program of calling the passbook interest rate by the name PBRATE wherever it is used. To change the rate, you would need only change the definition of the symbol at the beginning of the program.

On the other hand, if you had written the interest rate as an explicit constant in several places throughout the program, it might be difficult to make all the necessary changes.

Experienced programmers have learned that such modifications are the rule more often than the exception. Try to anticipate values that might change and represent them as symbols rather than as explicit constants.

Using symbols also improves program readability. Compare these two statements.

```
AREA = 3.1415965*RADIUS**2
AREA = PI*RADIUS**2
```

Using the symbol PI is preferable to using the constant 3.14159265. You can see at a glance that the second formula has the correct form. Did you notice that the first formula has an incorrect value for the constant π? Of course, the symbol PI must be assigned a value somewhere, as in the statement

```
PARAMETER (PI = 3.14159265)
```

But this is the only place in the program you have to look to verify that you used the correct value.

One other reason to use symbols for constants is that you can easily change the precision of the constant. Although the value of π will never change, you might decide to use an approximation of more or fewer decimal digits, depending, possibly, on the computer you are using. Incidentally, it is always acceptable to specify a constant with more precision than the computer can handle. If your computer provides seven-place precision for a real constant, you can still write a constant with more than seven significant digits. Many compilers will print a warning diagnostic to notify you of possible roundoff error.

STYLE MODULE—WRITING EXPRESSIONS

Careful typography in writing expressions will improve the readability of your programs.

Recall that blank spaces can be used almost anywhere in a FORTRAN statement. In fact, as long as the statement is typed in columns 7–72 of the line, FORTRAN completely ignores spaces except within a character constant. Thus, it makes no difference whether you write

```
X=(Y/2+3)
```

or

```
X = (Y/2 + 3)
```

Spaces are ignored even within variable names. If you like, you can write some variables as two words. For instance

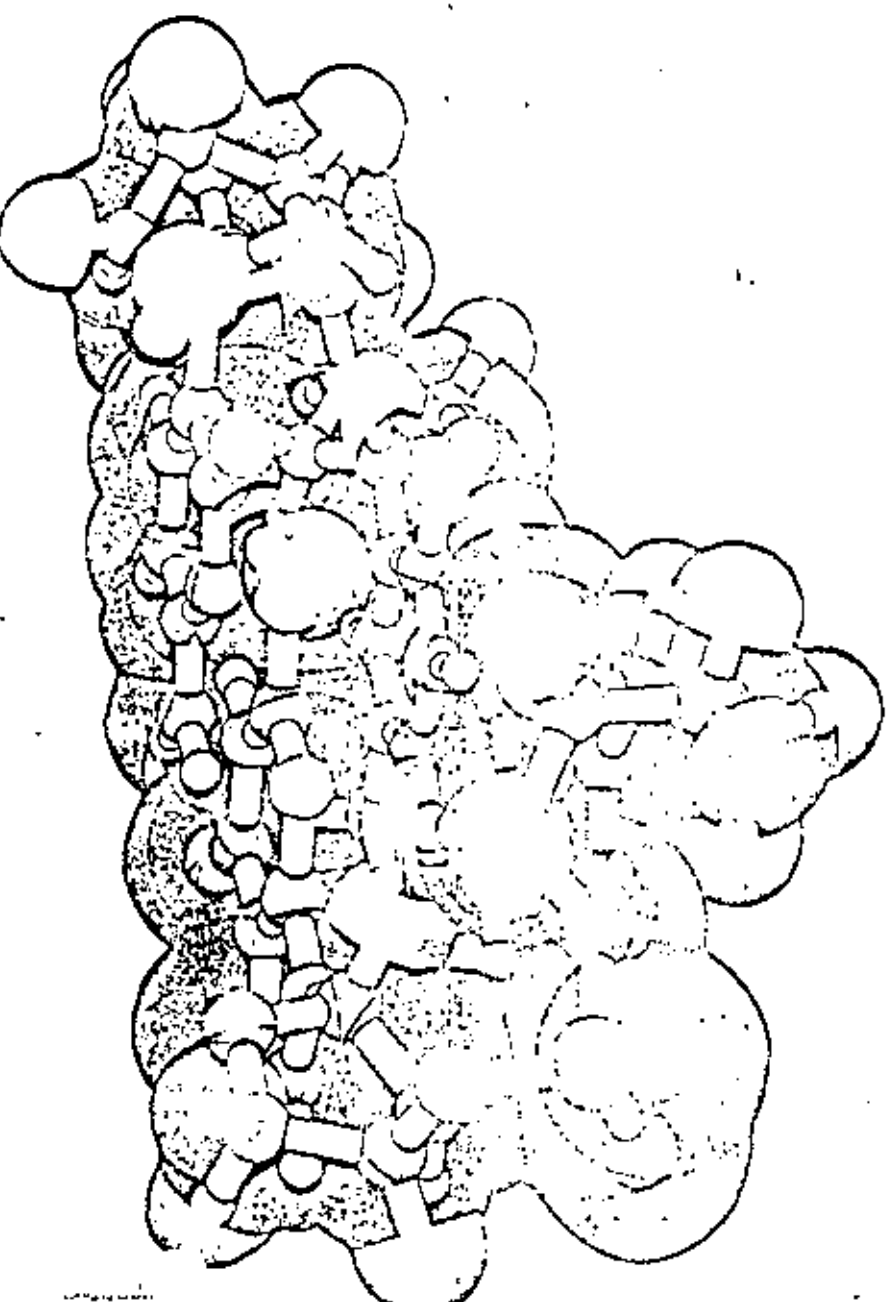
```
X VALUE = 0.5 * Y VALUE
PRINT *, FED TAX, SUM 1
```

Do not be deceived by the spaces; the variables are XVALUE, YVALUE, FEDTAX and SUM1.

Write all expressions in the form that is easiest for you to read. Never sacrifice clarity in an attempt to make a minor gain in efficiency.

Consider these two ways of computing the volume of a sphere.

```
* METHOD 1
      VOLUME = 4.18879021*R**3
* METHOD 2
            PARAMETER (PI = 3.14159265)
      VOLUME = 4.0/3.0*PI*R**3
```



CONTROL STATEMENTS AND STRUCTURED PROGRAMMING

INTRODUCTION One of the remarkable things about a computer is its ability to make decisions. Some computer programs make decisions so complex that the programs seem to exhibit almost human intelligence. Yet all programs, even the most complex, are made up of computations and elementary decisions like the ones you studied in Chapter 1. These elementary decisions enable a program to choose between alternative instruction sequences depending on the outcome of a single comparison of two quantities.

In FORTRAN, you program a decision using an IF statement. You were introduced to one form of the IF in Chapter 1. In this chapter you will learn about other forms of the IF, and about the related ELSE IF and ELSE statements.

An important use of decisions is in controlling program loops. One way of writing a loop is to use an IF statement in conjunction with a GO TO. Another method is provided by the DO statement.

This chapter also covers the other FORTRAN statements used to control the execution of a program: the STOP, PAUSE, and GO TO statements.

A main theme of this chapter is the need for structured programming. This chapter explains the basic control structures for writing program loops and decisions. Structured programming is a method of combining these control structures in an organized way that clearly reflects the program's meaning. The method is very important for writing programs that are easy to understand, and therefore reliable.

CONTROL STRUCTURES

The ...uctured flowcharts we have been using are made up of rectang... boxes containing basic instructions. These boxes are assembled to show the order in which

instructions are carried out. The various ways of assembling the boxes represent the different control structures used in expressing the logic of an algorithm.

We define a **basic block** in a structured flowchart to be either:

1. An empty box, or
2. A box containing an instruction that does not transfer control (for example, an assignment, or an output instruction)

Consider now how basic blocks can be combined in a structured flowchart. We can define a **block** in a structured flowchart through an indirect definition. A *block* in a structured flowchart is either:

1. A basic block, or
2. Any combination of basic blocks that can be constructed from the four rules shown in Figure 4.1

These four rules for constructing a block are the four basic control structures:

1. Sequential structure
2. If-then-else structure
3. While-loop structure
4. Repeat-loop structure

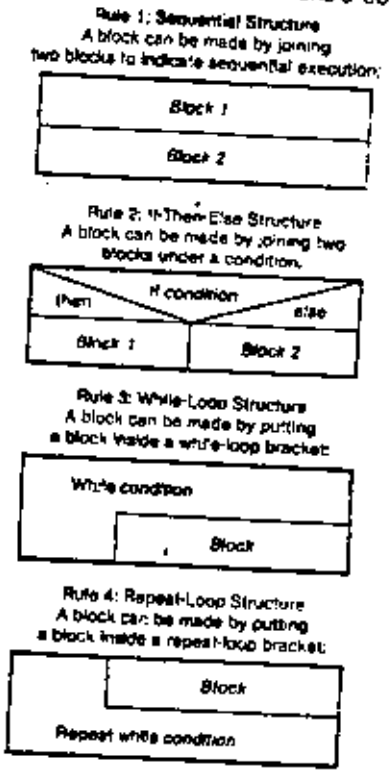
It may help you to understand the definition of a block if you think of it as a sort of puzzle. Suppose you have an assortment of boxes of various shapes and sizes cut out of paper. You can put them together, according to any of the four rules, to make a block. You can put one or more such blocks together, again according to any of the four rules, to make other blocks, and so on. The object of the puzzle is to make structured flowcharts. Figure 4.2 shows a specific example. The structured flowchart at the right of the figure is made in six steps by applying the rules.

You can try this puzzle for yourself. Draw any structured flowchart you can think of, then try to construct it from basic blocks by applying the rules, as in Figure 4.1. With a little experimentation, you can convince yourself of a basic fact: Any structured flowchart can be constructed in this manner. To be more precise, we can state this fact as follows:

> ***Principle of Structured Programming***
> Any algorithm can be represented by a structured flowchart constructed from basic blocks according to the four rules listed in Figure 4.1.

One way of looking at this principle is that it provides a definition of a structured flowchart: A *structured flowchart* is the same thing as a *block*, as previously defined. But the real significance of the principle is that four control structures are all we need. Any program, no matter how long or how complex, can be written by using these four

FIGURE 4.1
FOUR BASIC CONTROL STRUCTURES

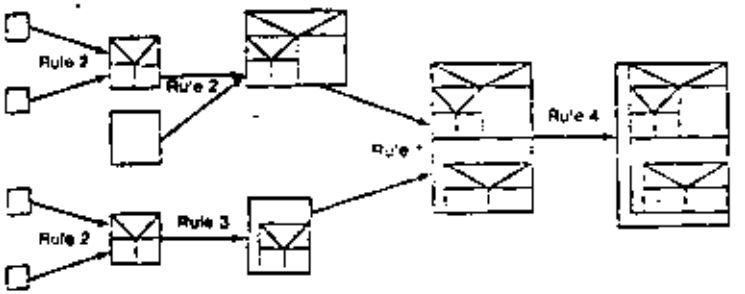


control structures in an organized way. Later in this chapter we will reexamine this principle to see how it applies to writing structured programs in FORTRAN.

THE GO TO STATEMENT

The **GO TO statement** transfers control to another statement in the program. To specify which statement is to be performed next, you use a statement label. A statement label (sometimes called a statement number) is just a one- to five-digit number written in columns 1 through 5 of the line. Figure 2.5 on page 38 gives

FIGURE 4.2



Any structured flowchart can be constructed by applying four rules which correspond to the basic control structures

examples of statement labels. The form of the GO TO statement is

```
GO TO n
```

where *n* is a statement label. This causes program control to transfer to the statement with label *n*.

Any statement can have a label. However, no two statements in a program can have the same label. Also, although a nonexecutable statement (such as INTEGER) may have a label, a control statement must not transfer to a nonexecutable statement.

STYLE MODULE—USING STATEMENT LABELS

It will improve the readability of your programs if you use statement labels in increasing order. Statement 20 should come after statement 10. Statement 30 should come after statement 20. This is not required in FORTRAN, but it will help you to find your way through your program. When you read a statement that says GO TO 100, you will know whether to look down the page or up the page for statement 100.

It is a good idea also to skip some numbers between statement labels. Label your statements 10, 20, 30, rather than 1, 2, 3. That way, if you modify your program, you can insert new statement labels and still keep them in increasing order.

STOP, PAUSE, AND END STATEMENTS

The FORTRAN statement

```
STOP
```


simply tells the machine to stop executing your program. This STOP statement is an executable statement which can appear at any point in your program.

The END statement also has a simple form:

```
END
```

Every program must contain one and only one END statement, which must be the last line of the program.

The END statement serves a dual purpose. It is an executable statement that tells the machine to stop running; in this respect it is identical to the STOP statement. But the END statement is also a declarative statement that marks the end of your FORTRAN program, so that the FORTRAN compiler knows when it has read all your program cards.

The following program contains both a STOP and an END statement:

```
REAL X
READ *, X
IF (X .EQ. 0) THEN
    PRINT *, 'THE VALUE IS ZERO'
    STOP
END IF
PRINT *, 'THE VALUE IS NOT ZERO'
END
```

If the number read is zero, the STOP statement terminates execution. Otherwise, the END statement terminates execution.

A STOP statement can include a character string in quotes, such as

```
STOP 'PROGRAM FINISHED'
```

The character string may or may not be used for anything, depending on your particular computer system. On some systems, the character string is printed at the end of your output when the STOP statement is executed. This feature is useful for debugging—especially if you have several different STOP statements in the program.

The PAUSE statement serves almost exactly the same purpose as the STOP statement. The form is

```
PAUSE
```

or

```
PAUSE 'character string'
```

The effect of this statement is like a STOP, except that after a PAUSE statement, the computer operator can manually restart the program, which then continues at the statement following the PAUSE.

THE IF-THEN-ELSE STATEVENTS

To decide between two paths in a program you can use an IF-THEN statement (also called a block IF statement) along with an ELSE statement and an END IF statement. Figure 4.6 shows an example. In the figure, *condition* represents some condition that is either true or false, while *Block 1* and *Block 2* represent any FORTRAN statement or group of FORTRAN statements. If the condition is true, the program executes *Block 1*; if the condition is false, the program executes *Block 2*. Figure 4.7 shows a specific example which corresponds to the following FORTRAN statements.

```
IF (N .EQ. 0) THEN
    PRINT *, 'ERROR: N WAS ZERO'
ELSE
    PRINT *, 'NUMBER OF VALUES = ', N
    PRINT *, 'SUM = ', SUM
    PRINT *, 'AVERAGE = ', SUM/N
END IF
```

The ELSE and END IF statements, by themselves, do nothing. Technically, they are executable statements, but they cause no machine action when executed. They serve only to delimit the IF block and the ELSE block, which are the two groups of statements between which the IF statement chooses.

To be precise, the IF block consists of all the executable statements following a block IF statement up to, but not including, the ELSE or END IF statement that goes with the block IF. The ELSE block consists of all the executable statements following the ELSE statement up to, but not including, the END IF statement that goes with the ELSE.

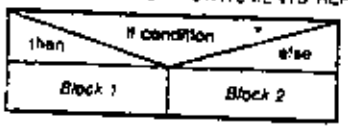
The only legal way to enter an IF block or an ELSE block is by executing the block IF or ELSE statement.

> ***Rule for Transfer Into a Block***
> A program may not transfer into an IF block or an ELSE block (or an ELSE IF block, covered later) from outside the block.

This rule implies that the following statements are not allowed.

```
* ILLEGAL TRANSFER INTO AN IF BLOCK
      IF (N .EQ. 0) THEN
          PRINT *, X
100       Y = X**2
          PRINT *, Y
      END IF
      GO TO 100
```


FIGURE 4.6
THE IF, ELSE, AND END IF STATEMENTS REPRESENT A DECISION




```
IF (condition) THEN
    Block 1
ELSE
    Block 2
END IF
```

The GO TO statement is illegal because it transfers control from outside an IF block to inside the block. Note that the rule does not forbid branching *out of* an IF block, nor does it forbid branching from one point to another *within* an IF block.

RELATIONAL EXPRESSIONS

Within the parentheses of an IF statement you write a condition, such as

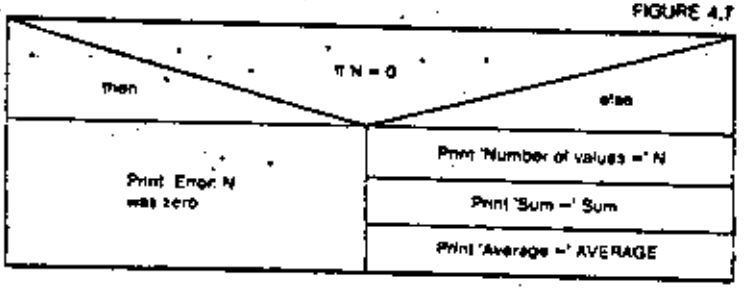
```
A .GT. 10
```

This kind of condition is called a **relational expression**. The symbol

```
.GT.
```

is called a **relational operator**. There are six such operators in FORTRAN. Table 4.1 lists them. You can use these operators to compare any two arithmetic expressions.

FIGURE 4.7

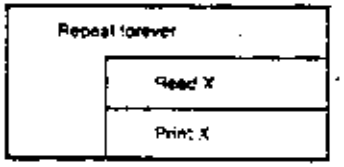


The exact effect of the PAUSE statement depends on the particular computer system you use. On a timesharing system, it may suspend execution of your program and wait for you to type a command to continue the program execution or end it. This statement can be useful for debugging on a timesharing system.

THE END-OF-FILE SPECIFIER IN THE READ STATEMENT

Figure 4.3 shows a procedure to read a number from a card then print the number and repeat the process.

FIGURE 4.3
A NEVER-ENDING LOOP



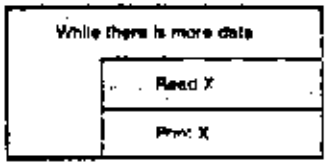
You can write this procedure with three FORTRAN statements.

```
10   READ *, X
     PRINT *, X
     GO TO 10
```

After reading each number and printing it, the GO TO statement branches back to the beginning of the loop. This will work fine—as long as there are more data cards to read. However, after the last card is read, the READ statement has nothing to read. This will cause an error condition, and the program will stop after printing some error message.

It would be better to have the program determine automatically that there are no more data cards to read, as in Figure 4.4.

FIGURE 4.4
A LOOP THAT EXITS WHEN ALL THE DATA HAS BEEN READ



To do this in FORTRAN, you can include an end-of-file specifier in the READ statement. An example is:

```
READ (*, *, END = 20) X
```

The symbol

```
END = 20
```

is the end-of-file specifier. This READ statement is like a combination READ and GO TO statement. It means to read a value for X, *but* if there is no more input data, to go to statement number 20. Using this form of the READ, you can write the procedure in Figure 4.4 like this:

```
10   READ (*, *, END = 20) X
     PRINT *, X
     GO TO 10
20   END
```

If there are 10 data cards to be read, the end-of-file specifier has no effect the first 10 times through the loop. The program just reads a value for X, and then goes on to the PRINT statement. The eleventh time the READ is executed, there is no more data, so the program branches to statement 20, the END statement, which stops execution.

The general form of the READ statement with an end-of-file specifier is

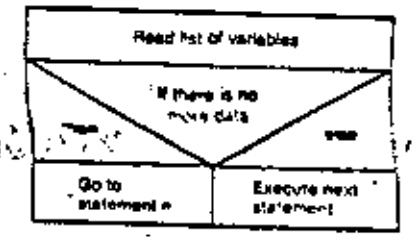
```
READ (*, *, END = n) list
```

where *n* is a statement label, and *list* is the list of variables to read. It means: "Read values for the variables, but branch to statement *n* if there is no more data to read."

In a structured flowchart, you can represent the end-of-file specifier in the looping condition as in Figure 4.4, or by a decision as in Figure 4.5.

FIGURE 4.5
ANOTHER WAY TO DIAGRAM THE END-OF-FILE SPECIFIER

READ (*, *, END = *n*) *list of variables*



For example,

```
N + 1 .LT. 2*A + 3*B
```

means "Is N + 1 less than 2A + 3B?" In algebra, you would write this as:

$$(N + 1) < (2A + 3B)$$

TABLE 4.1
RELATIONAL OPERATORS

FORTRAN Symbol	*Mathematical Symbol*	*Meaning*
.LT.	$<$	Less than
.LE.	$\leq$	Less than or equal to
.GT.	$>$	Greater than
.GE.	$\geq$	Greater than or equal to
.EQ.	$=$	Equal to
.NE.	$\neq$	Not equal to

If you compare two expressions of different types, a type conversion takes place, just as in mixed-mode arithmetic. For example, suppose that N is an integer variable and X is a real variable. Then to evaluate the relational expression

```
N .GE. X
```

FORTRAN converts N to a real value, subtracts the real value of X, and compares the result to zero.

All of the examples of IF statements that you have seen thus far have used simple relational expressions. The condition in an IF can be any relational expression, or, in fact, any *logical expression*.

LOGICAL EXPRESSIONS

Relational expressions are one kind of logical expression. A logical expression in FORTRAN is an expression that is either *true* or *false*. Such expressions are called *logical* because they are like the expressions used in the mathematical study of symbolic logic.*

The two simplest logical expressions in FORTRAN are the two logical constants, which represent the values *true* and *false*. In FORTRAN, you write these constants as:

*Symbolic logic, in its modern form, was invented by George Boole in his *Laws of Thought* (1854). Symbolic logic is sometimes called Boolean logic, and logical expressions are sometimes called *Boolean expressions*.


```
.TRUE.
```

and

```
.FALSE.
```

The periods around each of these words are part of the symbol for the logical constants.

Another kind of logical expression is a logical variable. Just as an integer variable takes on integer values, a logical variable takes on logical values. Thus there are only two possible values for a logical variable, .TRUE. and .FALSE..

You can declare certain variables to be logical by using a LOGICAL statement, like the following:

```
LOGICAL P, Q
```

This statement means that P and Q are logical variables in the program.

The LOGICAL statement (like the INTEGER and REAL statements) is a declarative statement that must come before the first executable statement in the program.

You can assign a value to a logical variable using a logical assignment statement of the form:

```
variable = expression
```

where *variable* is a logical variable and *expression* is a logical expression. For example,

```
LOGICAL YES
YES = .TRUE.
```

Logical operators in FORTRAN join logical expressions together to form new logical expressions. Suppose you want to know if the value of the real variable X is between 1 and 10. In algebra you would write this condition as

$$1 \leq X \leq 10$$

In FORTRAN, you use the logical operator

```
.AND.
```

to join two relational expressions, as follows:

```
1 .LE. X .AND. X .LE. 10
```

The machine computes the value of this logical expression to be either .TRUE. or .FALSE.. It is true if and only if both of the relations are true.

The logical operator

.OR.

tests whether at least one of two logical expressions is true. The logical expression

N .LT. 1 .OR. N .GT. MAX

has the value .TRUE. if N is less than 1 or if N is greater than MAX. In everyday speech, the word "or" sometimes means "one or the other, but not both." If a menu says, "soup or salad," it usually means you cannot have both. But in FORTRAN, the .OR. operator means "one or the other, or possibly both."

The operator

.NOT.

gives a logical expression its opposite value. You write this operator in front of an expression to change its value from true to false, or vice versa. For example,

.NOT. I .LT. J

is true if I is not less than J.

The logical operator

.EQV.

tests whether two logical expressions are equivalent; that is, whether they have the same value.

P .EQV. Q

is true if P and Q are both true or both false.

The logical operator

.NEQV.

is the opposite of the .EQV. operator. It tests whether two logical expressions are *not* equivalent. The expression

P .NEQV. Q

is true if P is true and Q is false, or if P is false and Q is true. The .NEQV. operator is sometimes called an exclusive or operator because another interpretation of it is that

P .NEQV. Q

is true if P is true or Q is true but *not* both. Thus, it is like .OR., but it *excludes* the case where both are true.

Table 4.2 is called a truth table. It shows all possible outcomes of applying any of the logical operators to two logical expressions P and Q.

TABLE 4.2
TRUTH TABLE FOR FORTRAN LOGICAL OPERATORS

P	Q	.NOT. P	P .OR. Q	P .AND. Q	P .EQV. Q	P .NEQV. Q
.TRUE.	.TRUE.	.FALSE.	.TRUE.	.TRUE.	.TRUE.	.FALSE.
.TRUE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.	.FALSE.	.TRUE.
.FALSE.	.TRUE.	.TRUE.	.TRUE.	.FALSE.	.FALSE.	.TRUE.
.FALSE.	.FALSE.	.TRUE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.

In Chapter 3, you saw the rules for evaluating arithmetic expressions. One of these rules said that multiplication is done before addition, unless otherwise specified by parentheses. There are similar rules for evaluating logical expressions.

Rules for Evaluating Logical Expressions

Parentheses can be used to group parts of a logical expression.

All arithmetic operators (+, −, *, /, **) are evaluated first, according to the rules in Chapter 3.

Relational operators (.LT., .GT., .LE., .GE., .EQ., .NE.) are evaluated before logical operators.

Unless otherwise grouped by parentheses, logical operators are evaluated in the following order:

1. .NOT.
2. .AND.
3. .OR.
4. .EQV. OR .NEQV.

Subject to the preceding rules, expressions are evaluated left to right.

The first rule means that you can always use parentheses if you are not sure how to write a logical expression. The other rules tell you what happens if you leave off the parentheses. For example, the rules imply that

.NOT. I + 1 .EQ. 5 .OR. N .EQ. 1

means the same as

(.NOT. ((I + 1) .EQ. 5)) .OR. (N .EQ. 1)

The most common use of logical expressions is in IF statements. When the IF is executed, the logical expression in parentheses is evaluated to yield a value of ".TRUE." or ".FALSE.". This value determines whether the IF statement then transfers control to the IF block or the ELSE block.

OTHER FORMS OF THE IF STATEMENT

Figure 4.8 illustrates a procedure that says, "If N is less than 0, print the word 'ERROR' and set N equal to 0; otherwise, do nothing." You can write this in FORTRAN as follows:

```
IF (N .LT. 0) THEN
    PRINT *, 'ERROR'
    N = 0
ELSE
END IF
```

The ELSE block in this example is empty. In this case, you can omit the ELSE statement entirely, and write the procedure as

```
IF (N .LT. 0) THEN
    PRINT *, 'ERROR'
    N = 0
END IF
```

Note that the END IF statement is still needed to terminate the IF block.

It is also allowable to have an empty IF block. For example, Figure 4.9 shows a procedure that says, "If XMIN ≤ X ≤ XMAX, do nothing; otherwise print an error message and stop." You can write this in FORTRAN as follows.

```
IF (XMIN .LE. X .AND. X .LE. XMAX) THEN
ELSE
    PRINT *, 'ERROR, X OUT OF RANGE'
    STOP
END IF
```

Another way to write this is to reverse the condition and put the statements in the IF block.

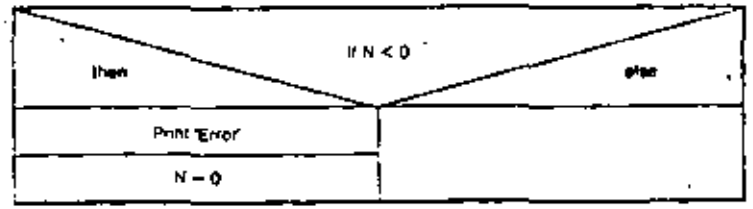
```
IF (.NOT. (XMIN .LE. X .AND. X .LE. XMAX)) THEN
    PRINT *, 'ERROR: X OUT OF RANGE'
    STOP
END IF
```

There is a short form of the IF statement which you can use to conditionally execute a single statement. An example is

```
IF (N LT. 100) N = N + 1
```

In place of the word "THEN" is the FORTRAN statement "N = N + 1." This assignment statement is executed if the condition "N .LT. 100" is true. Otherwise, the assignment statement is simply bypassed.

FIGURE 4.8
A CONDITIONAL STATEMENT WITH AN EMPTY ELSE BLOCK



This one-line IF statement is called a **logical IF statement.*** Figure 4.10 shows the general form. The condition in this statement can be any logical expression. The statement can be any executable statement except the following: DO, block IF, ELSE IF, END IF, END, or another logical IF. (Some of these are discussed later in this chapter.)

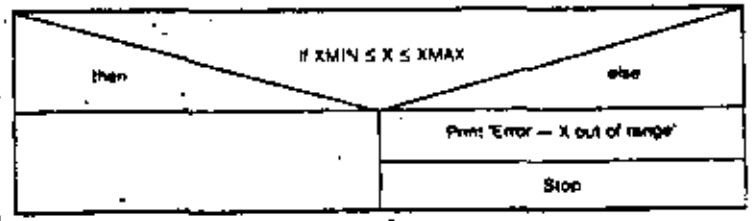
EXAMPLE 4.1 Percent Increase or Decrease

Dingles Drug Store has compiled sales figures for each item in their inventory. For each item, they prepared a data card giving:

1. The item number (an integer),
2. The dollar amount of sales for last month, and
3. The dollar amount of sales for this month

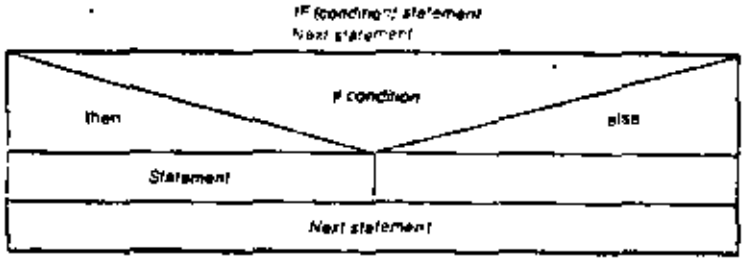
Last month's sales might be zero if it is a new item. This month's sales might be zero if it is a discontinued item. Write a program that will compute the percent increase (or decrease) in sales for each item. On each line of output, print the item name, the sales figures, and the percent increase followed by the words PERCENT INCREASE (or print the percent decrease, followed by the words PERCENT DECREASE).

FIGURE 4.9
A CONDITIONAL STATEMENT WITH AN EMPTY IF BLOCK



*"One-line IF statement" would be a better name for historical reasons, it is called the logical IF to distinguish it from the older arithmetic IF.

FIGURE 4.10



The logical (one-line) IF statement can select one statement to be executed if a condition is true.

Figure 4.11 shows a top-level design. To carry out this procedure, you can use three variables, ITEM, an integer variable for the item name, OLD AMT and NEW AMT, real variables for the sales amounts. To compute the percent increase in sales, you subtract the old amount from the new amount, then divide by the old amount. Multiply the result by 100 to give percent increase. The FORTRAN statement is:

```
PCT INC = 100*(NEW AMT - OLD AMT)/OLD AMT
```

There is one special case to watch out for. When OLD AMT is zero, the formula is invalid, because it calls for division by zero. You can use an ELSE statement to define the percent increase to be zero when OLD AMT is zero.

You can also use an ELSE statement to choose whether to print the words PERCENT INCREASE or PERCENT DECREASE.

Figure 4.12 shows the complete procedure. Here is the corresponding FORTRAN program.

```
* DINGLES DRUG STORE PROGRAM
* COMPUTE PERCENT INCREASE OR PERCENT DECREASE IN SALES
* FOR EACH ITEM.
      INTEGER ITEM
      REAL OLD AMT, NEW AMT, PCT INC
```

FIGURE 4.11
TOP-LEVEL DESIGN FOR THE PROGRAM IN EXAMPLE 4.1

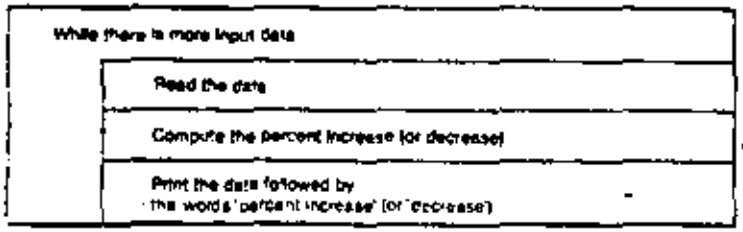
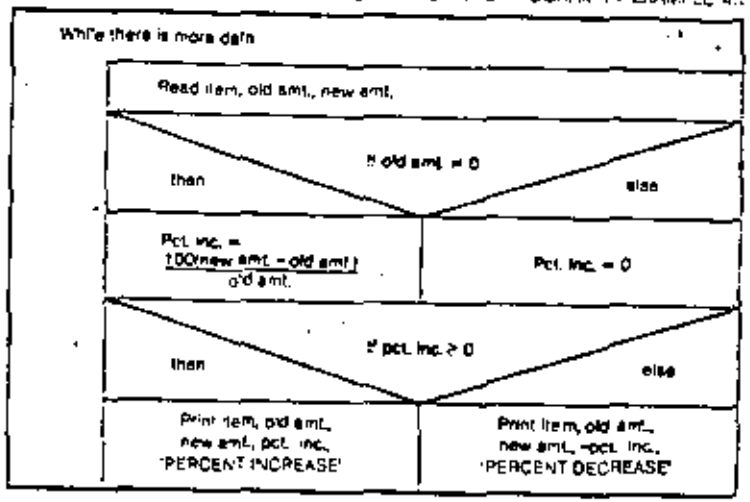


FIGURE 4.12
SECOND-LEVEL DESIGN FOR THE PROGRAM IN EXAMPLE 4.1




```
   10 READ (*, *, END = 900) ITEM, OLD AMT, NEW AMT
      IF (OLD AMT .NE. 0) THEN
         PCT INC = 100 * (NEW AMT - OLD AMT)/OLD AMT
      ELSE
         PCT INC = 0
      END IF
      IF (PCT INC .GE. 0) THEN
         PRINT *, ITEM, ' LAST MONTH: $', OLD AMT,
     $        ' THIS MONTH: $', NEW AMT, ' ',
     $        PCT INC, ' PERCENT INCREASE'
      ELSE
         PRINT *, ' LAST MONTH: $', OLD AMT,
     $        ' THIS MONTH: $', NEW AMT, ' ',
     $        PCT INC, ' PERCENT DECREASE'
      END IF
      GO TO 10
  900 END
```

STYLE MODULE—STRUCTURED PROGRAMMING

One of the greatest difficulties facing the beginning programmer is how to express program logic clearly. Without care, even a simple program can become a confused tangle of statements. Consider the following example:

```
      REAL X, LIMIT, DX, Y
      READ *, X, LIMIT, DX
 5    IF (X .LE. LIMIT) GO TO 15
      STOP
10    X = X + DX
      GO TO 5
15    Y = X**2 + X
      PRINT *, X, Y
      GO TO 10
      END
```

Each statement in this program is easy to understand. But what does the program do? It is difficult to tell. The procedure, though basically simple, is confusing because of the way it is written. See how much clearer the same procedure is when we rewrite it as follows:

```
      REAL X, Y, START, LIMIT, DX
      READ *, START, LIMIT, DX
      X = START
10    IF (X .LE. LIMIT) THEN
         Y = X**2 + X
         PRINT *, X, Y
         X = X + DX
         GO TO 10
      END IF
      END
```

Structured programming is a technique of building a program in an organized way by combining statements according to the basic control structures listed at the beginning of this chapter. We defined a *block* in a structured flowchart by using an indirect definition that gave rules for building blocks from other blocks. We can define a *program block* in a similar manner.

Definition of a Program Block
A program block is either:

1. A single executable statement that does not transfer control, or
2. Any combination of such statements that can be formed by the following rules.

Rule 1: Sequential structure. A program block can be made by joining two program blocks in sequence:

Block 1
Block 2

Rule 2: If-then-else structure. A program block can be made using the block IF, ELSE, and END IF statements in any of the following forms:

(a) IF (*condition*) THEN
  Block
  END IF

(b) IF (*condition*) THEN
  ELSE
  Block
  END IF

(c) IF (*condition*) THEN
  Block 1
  ELSE
  Block 2
  END IF

Rule 3: WHILE-loop structure. A program block can be made by putting a program block inside a while loop in the form:

label IF (*condition*) THEN
  Block
  GO TO *label*
  END IF

Rule 4: Repeat-loop structure. A program block can be made by putting a program block inside a repeat loop in the form:

label *Block*
  IF (*condition*) GO TO *label*

These rules are simply the FORTRAN version of the rules for making structured flowcharts. The principle of structured programming given earlier has the following counterpart for FORTRAN programs:

Principle of Structured Programming
Any algorithm can be written in FORTRAN by combining executable statements into a block according to the four rules given above.

We define a structured program to be a program that follows the rules of FORTRAN and in which all the executable statements form a block.

In Chapter 2 we said that *indentation* of statement groups can improve program readability. What we have actually been doing in our examples is indenting each structured program block. This practice lets you see at a glance just where each block in your program begins and ends. Thus, indentation of blocks is a typographical way of highlighting your program's control structure.

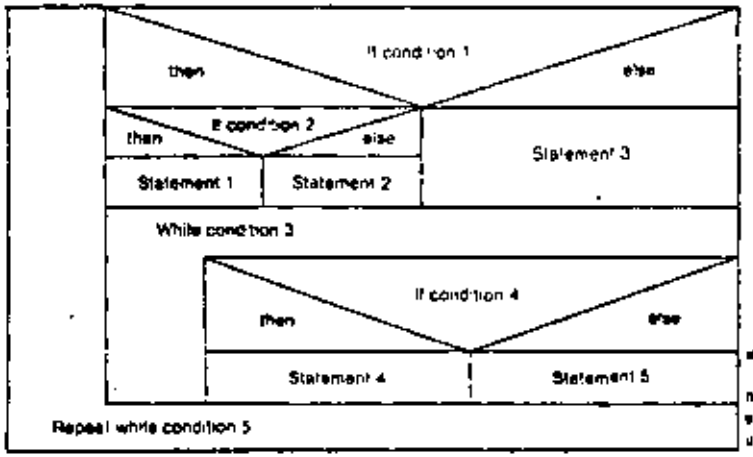
You may ask why structured programming is important. After all, if a computer carries out a set of instructions and comes up with the right answer, does it matter whether the instructions are written in structured form? The answer is that it certainly does not matter to the computer, and possibly does not matter to the end user of the output data, but it may matter a great deal to the programmer. Structured programs are by far easier to read and easier to understand than unstructured ones.

In a structured program, control flows from *top to bottom*. You can read a structured program like you read a book, starting at the beginning, ending at the end. Reading a highly unstructured program is like reading a book with a dozen footnotes per page. Your train of thought is constantly interrupted as your eyes dart about the page in seemingly random fashion. The logic behind the statements is obscure. A structured program is easier to read, and therefore much easier to validate. Validating a program—constructing an informal mental proof, or perhaps a formal written proof, of correctness—is ultimately the only way you have of knowing that the program works as you intended. Structured programming is your greatest asset in program validation.

EXAMPLE 4.2 Constructing a Structured Program

In the structured flowchart of Figure 4.13, *condition 1*, *condition 2*, and so on, can be any logical expressions, and *statement 1*, *statement 2*, and so on, can be any executable statements that do not transfer control. According to our definition, each of the statements are program blocks. Then by Rule 2, the following is a program block.

FIGURE 4.13




```
IF (condition 2) THEN
    statement 1
ELSE
    statement 2
END IF
```

By Rule 2 again, the following is a program block.

```
IF (condition 1) THEN
    IF (condition 2) THEN
        statement 1
    ELSE
        statement 2
    END IF
ELSE
    statement 3
END IF
```

In fact, you can continue to apply the rules for constructing a program block in the same way that the structured flowchart was built up in Figure 4.2 to show that the algorithm can be written as a structured program in the following form.

```
100  IF (condition 1) THEN
         IF (condition 2) THEN
             statement 1
         ELSE
             statement 2
         END IF
     ELSE
         statement 3
     END IF
200  IF (condition 3) THEN
         IF (condition 4) THEN
             statement 4
         ELSE
             statement 5
         END IF
         GO TO 200
     END IF
     IF (condition 5) GO TO 100
     END
```

Note that the form of this program—its structure—is the same regardless of the particular statements and conditions used.

Of course, to write a structured program you do not have to actually apply the rules in this mechanical fashion. With a little practice, you will learn to recognize the structured forms and use them automatically in your work. Your final result, though, is a program that could be built up step by step in the manner indicated by this example.

STYLE MODULE—LOOPS WITH AN EXIT IN THE MIDDLE

Consider the procedure in Figure 4.14. This is an example of a loop with an exit in the middle. The loop contains an instruction that exits if *condition 2* is true.

At first glance, this algorithm does not appear to fit into any of the patterns we covered; it is neither a while-loop structure nor a repeat-loop structure. We can, however, transform the algorithm into a while-loop structure by the technique of introducing a control variable.

Suppose that P is a logical variable which is not used anywhere else in the algorithm. To begin with, let P be true. Then modify the loop control condition to execute the loop while

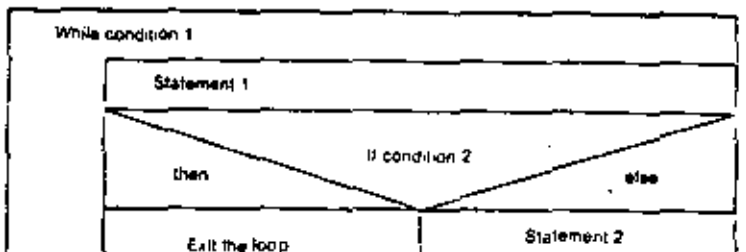
```
condition 1 .AND. P
```

is true. Within the loop, if *condition 2* is true, let P be false. This makes the loop control condition false, so that the loop terminates on the next iteration (Figure 4.15).

You should trace the logic of the algorithms in Figures 4.14 and 4.15 to convince yourself that they are equivalent.

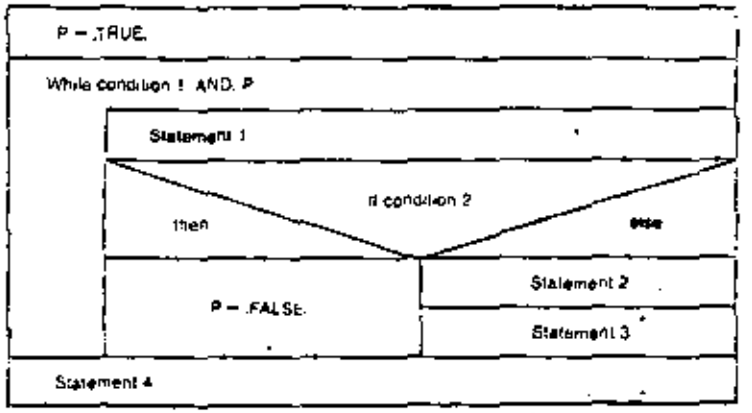
A similar technique can be used to transform a repeat loop with an exit in the middle into a repeat loop with no such exit (Exercise 17). In fact, a similar technique

FIGURE 4.14
A LOOP WITH AN EXIT IN THE MIDDLE




```
10    IF (condition 1) THEN
          Statement 1
          IF (condition 2) GO TO 20
          Statement 2
          Statement 3
          GO TO 10
      END IF
```

FIGURE 4.15



Using a logical variable P, the procedure in Figure 4.14 can be written so that the exit condition is incorporated into the looping condition.

```
      P = .TRUE.
10    IF (condition 1 .AND. P) THEN
          Statement 1
          IF (condition 2) THEN
              P = .FALSE.
          ELSE
              Statement 2
              Statement 3
          END IF
          GO TO 10
      END IF
      Statement 4
```

101

can be used to transform a while loop into a repeat loop, or vice versa (Exercise 18).

Another kind of loop with an exit in the middle is a loop that terminates when end of data is reached on a READ operation. This case can also be handled by introducing a control variable, using the IOSTAT specifier, which is discussed in Chapter 9. It is often more convenient in FORTRAN to use an end-of-file specifier that branches to the end of the block. A good way to organize such procedures is to make the READ the first statement of a block and have the end-of-file specifier branch to the first statement of the following block. You can use the form:

```
label 1   READ (..., END = label 2) list
          Block
          GO TO label 1
label 2   Next Block
```

For all practical purposes, this can be considered a while loop, since the block is executed while there is more input data

EXAMPLE 4.3 Quadratic Equations

A *quadratic equation* is an equation of the form

$$ax^2 + bx + c = 0$$

where a, b, and c are constants, and x is a variable. In algebra, you learn that this equation has two solutions, x_1 and x_2, given by the formulas

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

$$x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

For example, you can solve the equation

$$x^2 - 3x + 2 = 0$$

by applying the above formula to find $x_1 = 2$, $x_2 = 1$. Let us write a program to read three numbers, a, b, c, and print out solutions to the corresponding quadratic equation.

To make the program convenient to use, we can have it read a set of data cards, where each data card contains three values: a, b, and c. For each data card, the program determines the corresponding solutions and prints them. Thus, the top-level design is that shown in Figure 4.16.

To make the program work correctly for all possible values of a, b, and c, there are several special cases to consider.

If a is zero, the equation is not really a quadratic equation; it is a linear equation,

$$bx + c = 0$$

which we must solve by a different formula. Thus, the top-level design can be expanded as shown in Figure 4.17.

The solution of the linear equation is

$$x = -\frac{c}{b}$$

This formula, however, does not work if $b = 0$, in which case the equation to be solved has the form:

$$0x = c$$

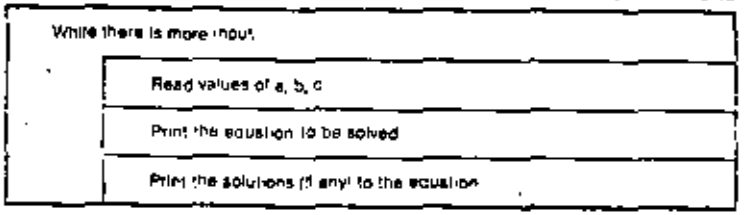
This equation is true for all x if $c = 0$ (the trivial case), and true for no x if $c \neq 0$ (the impossible case). Thus, the procedure for solving the linear equation is as shown in Figure 4.18.

The solution of the quadratic case ($a \neq 0$) is given by the formulas, but again there are three separate cases. The quantity under the square root sign

$$b^2 - 4ac$$

is called the *discriminant* of the equation. If $b^2 - 4ac > 0$, then the equation has two distinct

FIGURE 4.16
TOP-LEVEL DESIGN FOR SOLVING QUADRATIC EQUATIONS



solutions given by the formulas. If $b^2 - 4ac = 0$, then the equation has one solution given by

$$x = -\frac{b}{2a}$$

If $b^2 - 4ac < 0$, the equation has two solutions, but they are *complex numbers*, of the form

$$x + yi \qquad x - yi$$

where i is the base of the imaginary numbers ($i = \sqrt{-1}$) and x and y are given by the formulas

$$x = -\frac{b}{2a}$$

$$y = \frac{\sqrt{-(b^2 - 4ac)}}{2a}$$

FIGURE 4.17
SECOND-LEVEL DESIGN FOR SOLVING QUADRATIC EQUATIONS

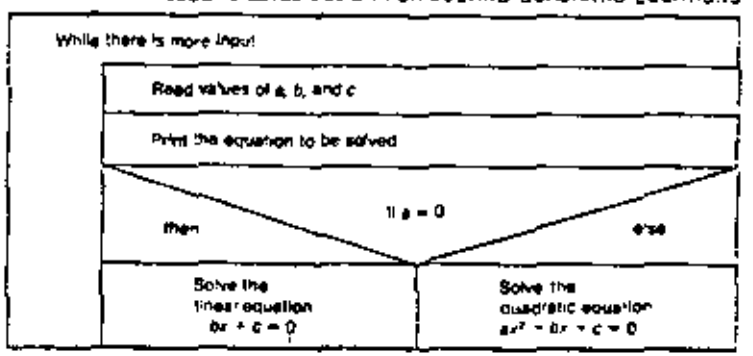
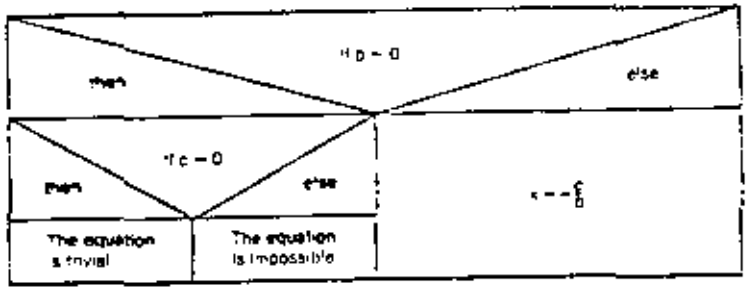


FIGURE 4.18
SOLUTION OF THE LINEAR CASE ($a \neq 0$)



Thus, the procedure for solving the quadratic case is as shown in Figure 4.19. Putting these parts together, we have the complete structured flowchart shown in Figure 4.20.

You can translate this procedure into FORTRAN using the structured forms previously described. There is only one point which we have not yet covered: finding square roots. The square root of a nonnegative value X is the same as X raised to the power 1/2. Thus, the square root of DISC is given by the FORTRAN expression

```
DISC**0.5
```

The complete program is as follows.

```
C  SOLVE THE QUADRATIC EQUATION
C
C     A*X**2 + B*X + C = 0
C
C  FOR ANY VALUES OF A, B, AND C.
C
C  THIS PROGRAM READS ANY NUMBER OF DATA CARDS. EACH DATA CARD
C  CONTAINS THREE NUMBERS AS VALUES OF A, B, AND C. THE PROGRAM
C  PRINTS EACH SET OF INPUT VALUES ALONG WITH THE SOLUTION TO THE
C  CORRESPONDING EQUATION.
C
      REAL A, B, C, DISC
10    READ (*, *, END = 90) A, B, C
      PRINT *, 'THE EQUATION'
      PRINT *, A, 'X**2 + ', B, 'X + ', C, ' = 0'
      IF (A .EQ. 0) THEN
          IF (B .EQ. 0) THEN
              IF (C .EQ. 0) THEN
                  PRINT *, 'IS TRIVIAL'
              ELSE
                  PRINT *, 'IS IMPOSSIBLE'
              END IF
```

FIGURE 4.19
SOLUTION OF THE QUADRATIC CASE ($a \neq 0$)

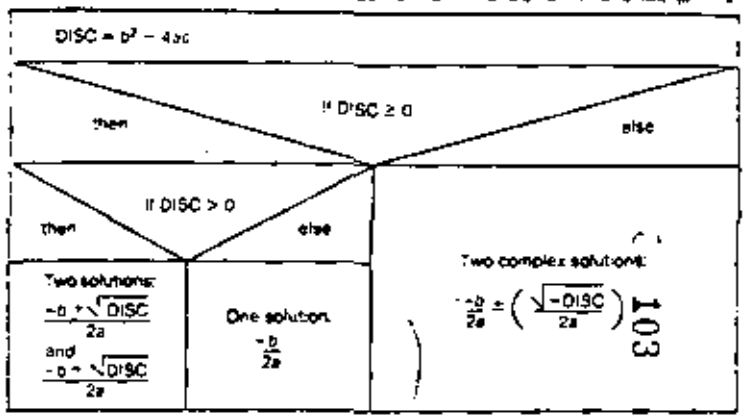
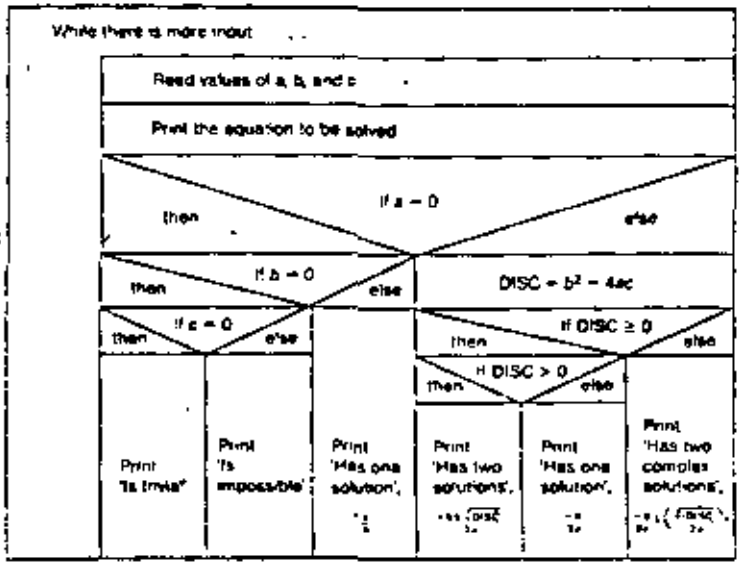


FIGURE 4.20
COMPLETE ALGORITHM FOR THE PROGRAM TO SOLVE QUADRATIC EQUATIONS



```
            ELSE
                PRINT *, 'HAS ONE SOLUTION'
                PRINT *, 'X = ', -C/B
            END IF
        ELSE
            DISC = B**2 - 4*A*C
            IF (DISC .GE. 0) THEN
                IF (DISC .GT. 0) THEN
                    PRINT *, 'HAS TWO SOLUTIONS'
                    PRINT *, 'X1 =',
     $                  (-B + DISC**0.5)/(2*A)
                    PRINT *, 'X2 =',
     $                  (-B - DISC**0.5)/(2*A)
                ELSE
                    PRINT *, 'HAS ONE SOLUTION'
                    PRINT *, 'X =', -B/(2*A)
                END IF
            ELSE
                PRINT *, 'HAS TWO COMPLEX SOLUTIONS'
                PRINT *, -B/(2*A), ' + OR - ',
     $              (-DISC)**0.5/(2*A), ' *I'
            END IF
        END IF
        GO TO 10
90      END
```

THE ELSE IF STATEMENT

Often you may need to use a whole series of conditions in a procedure. Figure 4.21 shows a procedure that executes one of four statements based on tests of three conditions. You can write this procedure in FORTRAN as follows:

```
IF (condition 1) THEN
    statement 1
ELSE
    IF (condition 2) THEN
        statement 2
    ELSE
        IF (condition 3) THEN
            statement 3
        ELSE
            statement 4
        END IF
    END IF
END IF
```


These statements are perfectly correct. There is, however, a more compact way of writing the procedure using the ELSE IF statement, as follows:

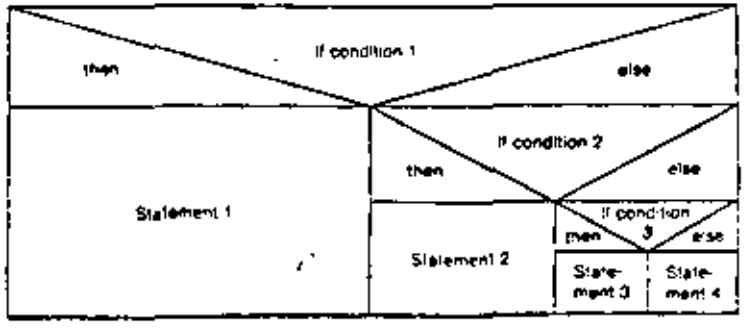
```
IF (condition 1) THEN
    statement 1
ELSE IF (condition 2) THEN
    statement 2
ELSE IF (condition 3) THEN
    statement 3
ELSE
    statement 4
END IF
```

This procedure means exactly the same thing as the one previously given. The first ELSE IF statement terminates the IF block and begins a new block, called an **ELSE IF block**. Thus, the ELSE IF combines the functions of an ELSE and an IF statement. Written this way, the procedure requires only one END IF statement, rather than three.

Using a structured flowchart, there is a more compact way of writing this kind of procedure. Figure 4.22 is a short way of writing the procedure in Figure 4.21. The instruction in the upper box means to test the conditions from left to right. If any condition is true, control passes to the instructions written under that condition.

Figure 4.23 shows a specific example of a procedure to select among one of three cases. As often happens in practice, the cases here are *mutually exclusive*—that is, exactly one of the conditions must be true. Thus, the order in which the conditions are tested does not matter, and you can diagram the selection of the case in the even simpler form shown in Figure 4.24. The corresponding FORTRAN statements are as follows:

FIGURE 4.21
CHAIN OF DECISIONS TO SELECT AMONG FOUR ALTERNATIVES



```
READ *, N
IF (N .LT. 0) THEN
    PRINT *, 'THE NUMBER IS NEGATIVE'
ELSE IF (N .EQ. 0) THEN
    PRINT *, 'THE NUMBER IS ZERO'
ELSE
    PRINT *, 'THE NUMBER IS POSITIVE'
END IF
PRINT *, 'THE END'
END
```

EXAMPLE 4.4 Tax Computation

The following table shows a hypothetical table for figuring income tax.

If income is OVER	*But NOT OVER*	*Tax is*
$0	$4000	15% of income
$4000	$6000	$600 + 19% of excess over $4000
$6000	$10,000	$980 + 23% of excess over $6000
$10,000	$12,000	$1900 + 25% of excess over $10,000
$12,000	$15,000	$2400 + 32% of excess over $12,000
$15,000	$20,000	$3360 + 35% of excess over $15,000
$20,000	$30,000	$5110 + 38% of excess over $20,000
$30,000	—	$8910 + 45% of excess over $30,000

FIGURE 4.22
ANOTHER WAY OF WRITING A CHAIN OF DECISIONS

If Condition 1	Else If Condition 2	Else If Condition 3	Else
Statement 1	Statement 2	Statement 3	Statement 4

FIGURE 4.23
AN IF . . . ELSE IF . . . ELSE STRUCTURE TO SELECT ONE OF THREE CASES

Read N		
If N < 0	Else If N = 0	Else
Print 'The number is negative'	Print 'The number is zero'	Print 'The number is positive'

FIGURE 4.24
A SIMPLER WAY OF DIAGRAMMING THE PROCEDURE OF FIGURE 4.23

Read N		
Select case N < 0	N = 0	N > 0
Print 'The number is negative'	Print 'The number is zero'	Print 'The number is positive'

The taxpayer is allowed a $500 deduction from his or her net income for each dependent. A set of data cards contains data on taxpayers. The data on the card is

Social security number (written as three numbers with a space between them, that is, III 22 3333)
Number of dependents (an integer)
Net income, in dollars (a real number)
Amount withheld, in dollars (a real number)

Write a program to compute each person's tax and the refund due (or amount owed). Some sample output is

```
SOCIAL SECURITY NUMBER 111 - 22 - 3333
INCOME              $ 10000.0
DEDUCTIONS          2
TAXABLE INCOME      $ 9000.0
TAX                 $ 1670.0
AMT. WITHHELD       $ 1700.0
REFUND DUE          $ 30.0
```

Figure 4.25 shows a top-level design for the program.

FIGURE 4.25
TOP-LEVEL DESIGN FOR EXAMPLE 4.4

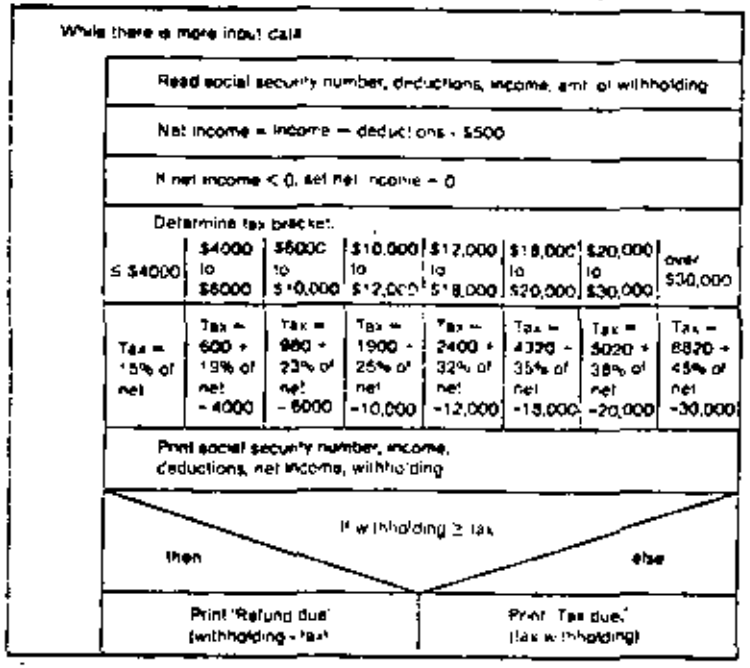
While there is more input data
Read data for one person
Compute the net income by subtracting $500 per deduction. If result is negative, use 0.
Determine the appropriate tax bracket and compute the tax
Print the output for this person

To determine which range the adjusted income falls in, you can use a series of ELSE IF statements. You can expand step 4 of the top-level outline as shown in Figure 4.26.

In the following FORTRAN program, we used symbolic constants for all the numbers in the tax table. CUT 1 is the first cutoff point, $4000. PCT 1 is the percentage rate in formula 1, AMT 2 is the constant amount in formula 2, and so on. Here is the complete program.

```
* INCOME TAX COMPUTATION
*
      REAL INCOME, NET INC, WTHOLD
      INTEGER DEDUC, SOC 1, SOC 2, SOC 3
      REAL CUT 1, CUT 2, CUT 3, CUT 4, CUT 5, CUT 6, CUT 7
      REAL PCT 1, PCT 2, PCT 3, PCT 4, PCT 5, PCT 6, PCT 7, PCT 8
      REAL AMT 2, AMT 3, AMT 4, AMT 5, AMT 6, AMT 7, AMT 8
      PARAMETER (CUT 1 = 4000, CUT 2 = 6000, CUT 3 = 10000,
     $    CUT 4 = 12000, CUT 5 = 18000, CUT 6 = 20000
     $    CUT 7 = 30000)
```

FIGURE 4.26
PROGRAM DESIGN FOR EXAMPLE 4.4




```
      PARAMETER (PCT 1 = 0.15, PCT 2 = 0.19, PCT 3 = 0.23,
     $    PCT 4 = 0.25, PCT 5 = 0.32, PCT 6 = 0.35,
     $    PCT 7 = 0.38, PCT 8 = 0.45)
      PARAMETER (AMT 2 = 600, AMT 3 = 980, AMT 4 = 1900,
     $    AMT 5 = 2400, AMT 6 = 4320, AMT 7 = 5020,
     $    AMT 8 = 6820)
*
* REPEAT THE FOLLOWING FOR EACH INPUT CARD.
*
100   READ (*, *, END = 900) SOC 1, SOC 2, SOC 3
     $    DEDUC, INCOME, WTHOLD
*
* COMPUTE THE NET INCOME BY SUBTRACTING $500 PER DEDUCTION.
* IF THE RESULT IS NEGATIVE, MAKE IT ZERO.
*
      NET INC = INCOME - 500-DEDUC
      IF (NET INC .LT. 0) NET INC = 0
*
* COMPUTE THE TAX
*
      IF (NET INC .LE. CUT 1) THEN
          TAX = PCT 1 * NET INC
      ELSE IF (NET INC .LE. CUT 2) THEN
          TAX = AMT 2 + PCT 2 * NET INC
      ELSE IF (NET INC .LE. CUT 3) THEN
          TAX = AMT 3 + PCT 3 * NET INC
      ELSE IF (NET INC .LE. CUT 4) THEN
          TAX = AMT 4 + PCT 4 * NET INC
      ELSE IF (NET INC .LE. CUT 5) THEN
          TAX = AMT 5 + PCT 5 * NET INC
      ELSE IF (NET INC .LE. CUT 6) THEN
          TAX = AMT 6 + PCT 6 * NET INC
      ELSE IF (NET INC .LE. CUT 7) THEN
          TAX = AMT 7 + PCT 7 * NET INC
      ELSE
          TAX = AMT 8 + PCT 8 * NET INC
      END IF
*
* PRINT THE RESULTS.
*
      PRINT *, 'SOCIAL SECURITY NUMBER '.
     $    SOC 1, ' - ', SOC 2, ' - ', SOC 3
      PRINT *, 'INCOME            $', INCOME
      PRINT *, 'DEDUCTIONS        ', DEDUC
      PRINT *, 'TAXABLE INCOME $', NET INC
      PRINT *, 'AMT WITHHELD   $', WTHOLD
      IF (WTHOLD .GE. TAX) THEN
          PRINT *, 'REFUND DUE            $', WTHOLD - TAX
```

```
      ELSE
         PRINT *, 'TAX DUE                $', TAX - WTHOLD
      END IF
      GO TO 100
  900 END
```

THE DO STATEMENT

One of the most common types of program loops is one in which a variable counts the number of times the loop has been executed. The DO statement provides an easy way to write this kind of loop. A DO loop is actually just a while-loop structure in which a counter variable, called the DO variable, is incremented or decremented automatically on each iteration. Thus, you can use a DO loop to form structured programs in the same way as you would use the equivalent while loop.

The following program uses a DO statement to print a table of squares.

```
      INTEGER X
      DO 10, X = 1, 20
   10    PRINT *, X, ' SQUARE = ', X**2
      END
```

This DO statement means to execute statement number 10, the PRINT statement, first with X = 1, then with X = 2, then with X = 3, and so on, until X = 20. Thus the PRINT statement executes 20 times, and the output includes the squares of the numbers 1 through 20. You can represent this procedure with the structured flowchart shown in Figure 4.27.

Figure 4.28 shows the parts of the DO statement. After the word DO comes a statement label. The comma following this label is optional. Next comes the name of the DO variable, then an equal sign, then two or three arithmetic expressions, separated by commas. These expressions are called the initial parameter, the terminal parameter, and the incrementation parameter.

The statement label after the word DO is the label of the terminal statement of the loop. All the statements following the DO, up to and including the terminal statement, form the range of the DO loop. A DO loop can consist of a single statement, as in our first example, or it can include many statements.

The DO variable may be any real or integer variable name. The value of this variable changes as the DO loop executes. To begin with, the value of the DO variable is set to the value of the initial parameter. When the loop has executed once, the value of the incrementation parameter is added to the DO variable, and the loop repeats, starting with the first statement after the DO statement. This process continues. On each iteration of the DO loop, the value of the incrementation parameter is added to the DO variable. When the value of the DO variable is greater than the value of the terminal parameter, the program exits the DO loop, branching to the first statement after the terminal statement of the DO loop.

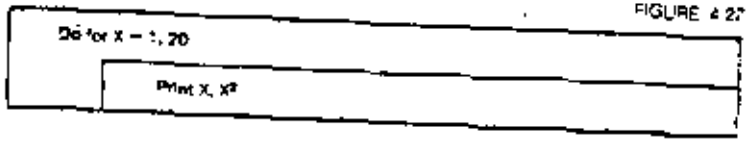


FIGURE 4.27

Each of the initial, terminal, and incrementation parameters may be a constant, or variable, or expression. For example,

```
DO 70, X = 1, 2*N + 1, INC/2
```

You can omit the incrementation parameter entirely (along with the comma before it). Omitting the incrementation parameter is the same as specifying a value of 1. Thus,

```
DO 100, I = 1, 20
```

means the same as

```
DO 100, I = 1, 20, 1
```

The incrementation parameter can be positive, or negative, but not zero. If it is negative, the DO variable counts "backward." For example,

```
DO 100, X = 50, 0, -1
```

means to execute the DO loop with X equal to 50, then 49, then 48, 47, and so on.

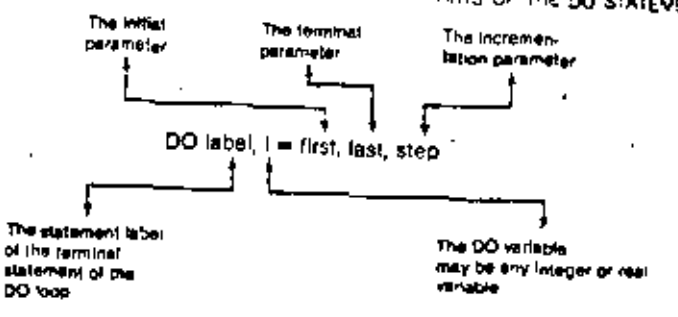


FIGURE 4.28
PARTS OF THE DO STATEMENT

The three parameters and the DO variable itself each may be of either integer or real type. For instance,

```
REAL X
DO 100, X = 0, 1.2, 0.1
```

means to execute the loop with X = 0.0, 0.1, 0.2, 0.3, . . . , 1.1, 1.2.

Let's see in more detail how the DO statement works. Consider the statement in its general form:

DO *label*, *var* = e_1, e_2, e_3

Executing the DO loop consists of nine steps.

Step 1: *Evaluate the parameters.* The values of the expressions e_1, e_2, and e_3 are computed. If the DO variable is an integer variable, the values of the parameters are converted to integer numbers. Similarly, if the DO variable is real, the values of the parameters are converted to real numbers. Remember that if you leave off the incrementation parameter, e_3, it is assumed to have the value 1.

Step 2: *Initialize the DO variable.* This has the effect of the assignment statement

$$var = e_1$$

Step 3: *Compute the iteration count.* The iteration count is the number of times the DO loop is to be executed. For example, the statement

```
DO 100, I = 1, 20
```

says to execute the DO loop 20 times, so the iteration count is 20. For the statement

```
DO 100, I = -10, 10
```

the iteration count is 21. For the statement

```
DO 100, I = 100, 0, -1
```

the iteration count is 101. In general, FORTRAN figures the iteration count using the formula

$$iteration\ count = (e_2 - e_1 + e_3)/e_3$$

If the result is not an integer, it is converted to an integer, truncating any fractional part. If the iteration count is less than zero, FORTRAN uses an iteration count of zero.

Step 4: *Test the iteration count.* If the count is zero, the program exits the DO loop, beginning with the statement following the terminal statement of the loop. If the count is positive, the program goes on to Step 5. Note that the iteration count might be zero the very first time it is tested, in which case the DO loop is not executed at all.

Step 5: *Execute the DO loop.* If there are no control statements in the loop, the program executes all the statements up to the terminal statement. If the loop contains control statements, it is possible that the program will exit the loop by branching to a statement outside the loop, in which case the DO loop is said to be inactive, and the loop processing is no longer controlled by these steps. For iteration to continue, the program must eventually go on to Step 6.

Step 6: *Execute the terminal statement.* Of course, if the DO loop consists of a single statement, then this is the same as Step 5. A more complicated DO loop may contain many statements, including control statements that branch within the loop. Whatever happens within the loop, the terminal statement must be executed before loop processing can continue.

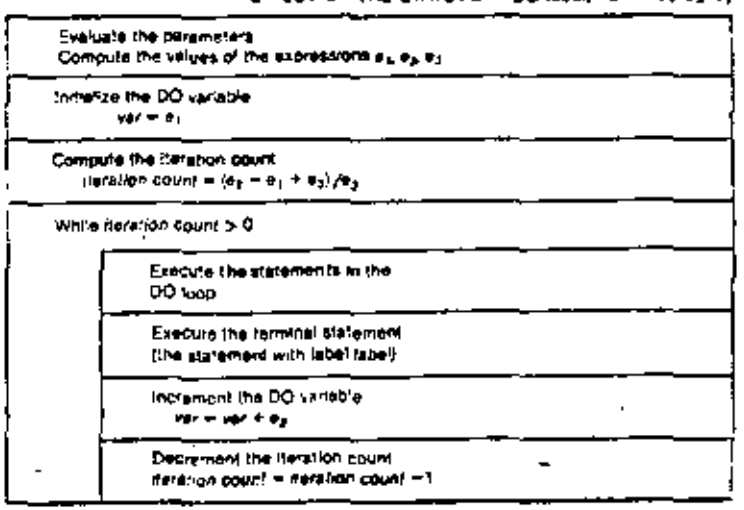
Step 7: *Increment the DO variable.* The value of the incrementation parameter is added to *var*.

Step 8: *Decrement the iteration count.* The new value of the iteration count is the old value minus 1.

Step 9: Go back to Step 4.

The effect of these nine steps is shown in Figure 4.29.

FIGURE 4.29
EFFECT OF THE STATEMENT DO *label*, *var* = e_1, e_2, e_3



THE CONTINUE STATEMENT

The **CONTINUE** statement has a simple form:

```
CONTINUE
```

This statement has a simple purpose, too. It does nothing. It is an executable statement that has no effect when it is executed.

As a matter of style, we generally use a CONTINUE statement as the terminal statement for a DO loop. For instance, the DO loop

```
      DO 20, I = 1, N
20        PRINT *, I
```

can also be written like this.

```
      DO 20, I = 1, N
          PRINT *, I
20        CONTINUE
```

These two DO loops have exactly the same effect.

As you will see in the following section, there are a few cases where a CONTINUE statement must be used to avoid certain restrictions of DO loops. In most cases, we just use it to make the program easier to read.

RULES FOR DO LOOPS

There are a few restrictions on the use of the DO statement, and a few points that need special attention.

> The value of the DO variable must not be redefined within the loop.

The value of the DO variable changes according to the incrementation procedure discussed earlier. You cannot circumvent that procedure by assigning a new value to the DO variable within the loop. Thus, the following loop is illegal.

```
* ILLEGAL DO LOOP
      DO 10, J = 1, L
          IF (J .EQ. K) J = J + 1
10        PRINT *, J
```

> A program must not branch into a DO loop from outside the loop.

This rule implies that the following program is illegal.

```
      DO 100, COUNT = 1, N + 1
50    PRINT *, COUNT
100   CONTINUE
* ILLEGAL GO TO STATEMENT.
      GO TO 50
```

The GO TO statement is *outside* the DO loop, but it branches to a statement *inside* the loop.

It is easy to see why this rule is necessary. If branching into a DO loop were allowed, then the values of the DO variable and the iteration count would be undefined at the end of the loop.

> Certain statements are not allowed as the terminal statement of a DO loop. They are:
>
> GO TO
> assigned GO TO
> arithmetic IF
> block IF
> DO
> ELSE IF
> ELSE
> END IF
> RETURN
> STOP
> END

A logical IF statement is allowed to be the terminal statement but only if it does not contain any of the above statements.

For completeness, this list of illegal terminal statements includes some statements we have not yet discussed.

This rule implies that the following DO loop is illegal.

```
* ILLEGAL DO LOOP
      DO 10, I = 1, L
10        IF (I*2 .EQ. J) GO TO 20
```

You can make this loop legal by using a CONTINUE statement, as follows.

```
* LEGAL DO LOOP
      DO 10, I = 1, L
          IF (I*2 .EQ. J) GO TO 20
10        CONTINUE
```

Although the CONTINUE statement itself does nothing, it is a legal terminal statement for a DO loop, whereas the IF statement is not.

DO loops and IF blocks, ELSE IF blocks, or ELSE blocks must not overlap. If a DO loop contains a block IF statement, the corresponding END IF statement must be within the DO loop. If a block IF and/or an ELSE IF, or an ELSE block appears within a DO loop, then the entire IF block must be inside the DO loop.

The following statements violate this rule.

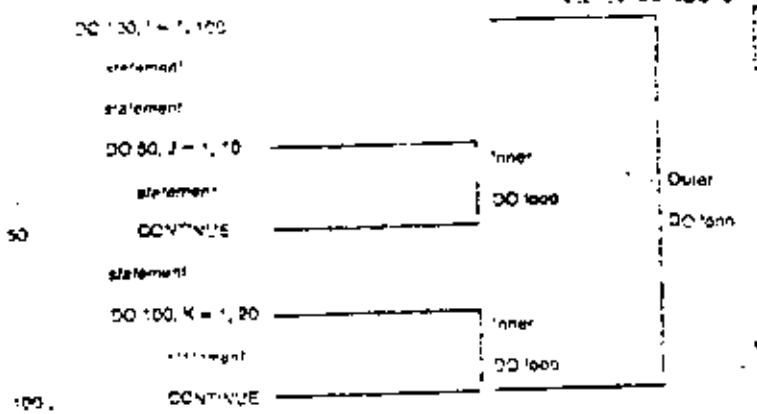
```
      ILLEGAL DO LOOP AND IF BLOCK
            IF (A .GT. B) THEN
               DO 10, I = 1, N
                  READ *, X
                  A = A - X
               END IF
10             CONTINUE
```

MORE RULES FOR DO LOOPS*

In this section we cover three rules that apply when you write one DO loop inside another DO loop, as illustrated in Figure 4.30. This arrangement is sometimes called nested DO loops. The inner DO loops are nested inside the outer one.

If two DO loops overlap, then one of them must be entirely inside the other.

FIGURE 4.30
NESTED DO LOOPS



*Optional section

According to this rule, the loops shown in Figure 4.31 are illegal. The rule does not, however, forbid two DO loops from having the same terminal statement. The third DO statement in this figure has the same terminal statement label (10) as the first DO statement in the figure.

Two or more DO loops may have the same terminal statement. Each time the terminal statement is executed, incrementation of the DO variable and testing of the iteration count are carried out for each loop, in turn, just as though each loop ended with a separate terminal statement.

In other words, the statements

```
      DO 20, I = 1, 10
      DO 20, J = 1, 10
20       PRINT *, I, J
```

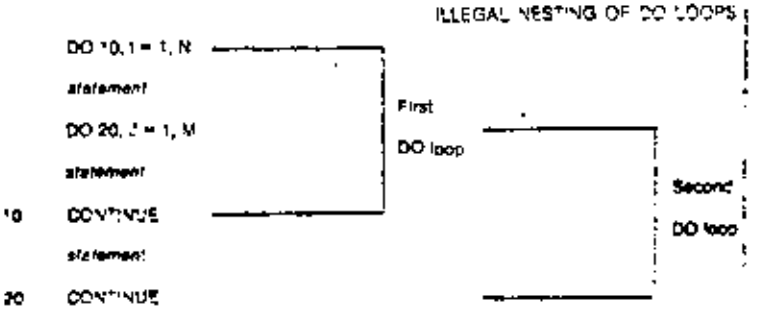
have the same effect as they would if written like this:

```
      DO 20, I = 1, 10
      DO 10, J = 1, 10
      PRINT *, I, J
10    CONTINUE
20    CONTINUE
```

The rules we gave earlier for DO loops apply to *each* loop in a set of nested DO loops. In particular, they forbid an inner DO loop from changing the value of the DO variable in an outer DO loop. To be specific:

Two nested DO loops may not use the same DO variable.

FIGURE 4.31
ILLEGAL NESTING OF DO LOOPS



Thus the following DO loops are *not* permitted.

```
      DO 20, I = 1, 10
      DO 10, I = 1, 10
      statement
10    CONTINUE
20    CONTINUE
```

STYLE MODULE—MISUSE OF THE DO STATEMENT

Not all loops are DO loops. The DO statement is useful for some kinds of loops, but sometimes you should use GO TO and IF statements instead.

Consider the following short program. This program is supposed to read a deck of cards, with one number on each card, and print the sum of all the numbers.

```
* POOR PROGRAM TO ADD NUMBERS
      REAL SUM, X
      INTEGER I, N
      SUM = 0
* READ THE NUMBER OF DATA CARDS
      READ, N

      DO 10, I = 1, N
         READ *, X
         SUM = SUM + X
10       CONTINUE
      PRINT *, N, ' CARDS WERE READ'
      PRINT *, 'THE SUM IS ', SUM
      END
```

This use of a DO loop makes things easy for the programmer, but it makes things hard for the user of the program. Suppose you wanted to add up the numbers on about 200 cards. You would have to count all the cards before you could run this program. And if you made a mistake in counting, the program would not work.

It is much better to let the computer do the counting. That is what computers are good at. Here is one way to do it.

```
* BETTER PROGRAM TO ADD NUMBERS
      REAL SUM, X
      INTEGER N, INFIN
      PARAMETER (INFIN = 50000)
      SUM = 0
      DO 10, N = 1, INFIN
         READ (*, *, END = 20) X
         SUM = SUM + X
10    CONTINUE
20    PRINT *, N - 1, ' CARDS WERE READ'
      PRINT *, ' THE SUM IS ', SUM
      END
```

This program uses a DO loop with a large upper limit. The upper limit was chosen to be so large that it would never be reached. The program should always exit the loop via the END = branch on the READ statement.

It is just as easy to write this program without using a DO loop at all. This is what we've done in the third version.

```
* IMPROVED PROGRAM TO ADD NUMBERS
      REAL SUM, X
      INTEGER N
      SUM = 0
      N = 0
10       READ (*, *, END = 20) X
         N = N + 1
         SUM = SUM + X
         GO TO 10
20    PRINT *, N, ' CARDS WERE READ'
      PRINT *, 'THE SUM IS ', SUM
      END
```

There are two lessons to be learned from this example, the first lesson is this:

> Use DO loops where appropriate. Use other kinds of loops where they are better suited to the program.

A DO loop is fine if you know in advance how many times the loop needs to be executed. But if you want a loop to repeat until some condition is true—for instance until the program runs out of data cards—then you should consider using a different control structure.

The second lesson is of a more general nature.

> Make your program easy to use.

Of course you want to make your program easy to write. That is fine. But do not do so at the expense of making it hard to use. For example, do not make the user count the data cards by hand. Computers have gotten a bad reputation over the years. Some people see them as inflexible, obstinate machines. In most cases, the programmer, not the computer, is to blame. The first program in this section is only a small example of the problem. But many large programs written by professional programmers are just as bad. As your skill in programming increases, you may very likely begin writing programs for others to use. As a programmer, whether student or professional, it is your responsibility to look out for the interests of the user.

STYLE MODULE—COMPARISON OF REAL EXPRESSIONS FOR EQUALITY

You can run into difficulty when you use a relational expression of the form

```
A .EQ. B
```

where A and B are real expressions. The following program illustrates the problem.

```
* TEMPERATURE CONVERSION TABLE
* (INCORRECT METHOD)
      REAL C, F
      F = 95
100   C = 5.0/9.0 * (F - 32)
      PRINT *, 'FAHRENHEIT:', F, ' CELSIUS:', C
      F = F + 0.1
      IF (F .EQ. 105.1) STOP
      GO TO 100
      END
```

This is a legal program, but on many computers it would get stuck in the loop. The reason is that F may never be equal to 105.1. Mathematically, F should be 105.1 on the 102nd iteration of the loop. But F is a real variable, and there is some inherent error in real variables because of the way they are stored in the computer. On the 102nd trip through the loop, F might be equal to 105.099999, which is very close to 105.1, but not exactly equal, so the loop repeats again, with F approximately equal to 105.2. The program might run forever unless stopped by the computer operator. (Usually, the computer operating system will automatically stop student jobs that run for more than a few seconds.)

To avoid this problem, observe the following practice:

> **Do not use relational expressions of the form**
>
> ```
> A .EQ. B
> ```
>
> **or**
>
> ```
> A .NE. B
> ```
>
> **where A or B is a real expression that might be approximate. Rewrite the program to use one of the forms**
>
> ```
> A .LT. B    A .LE. B
> A .GT. B    A .GE. B
> ```
>
> **or use a DO loop instead.**

STYLE MODULE—REAL DO VARIABLES

When you use a DO loop with a real DO variable, the value of the variable will be approximate. Consider the temperature conversion program in the previous example. We can rewrite this using a DO loop, as follows:

```
* TEMPERATURE CONVERSION PROGRAM
      REAL C, F
      DO 100, F = 95, 105, 0.1
         C = 5.0/9.0 * (F - 32)
         PRINT *, ' FAHRENHEIT:', F, ' CELSIUS:', C
100   CONTINUE
      END
```

This program is legal and it does not get stuck in a loop (because the DO loop tests the iteration count, not the value of F). But the output may look a bit strange. On a certain computer, the program might produce the following output.

```
FAHRENHEIT 95.00000     CELSIUS 32.00000
FAHRENHEIT 95.09999     CELSIUS 35.05555
FAHRENHEIT 95.19998     CELSIUS 35.11110
FAHRENHEIT 95.29997     CELSIUS 35.16665
. . .                   . . .
```

The programmer may have wanted the Fahrenheit temperatures to be exactly 95.0, 95.1, 95.2, 95.3, and so on. But there is a cumulative roundoff error that results from adding 0.1 to F on each iteration. So the values of F are slightly inaccurate and the output is hard to read.

The way around this problem is to use integer arithmetic. It is less convenient to use, but it is more precise, and in some applications, that is important.

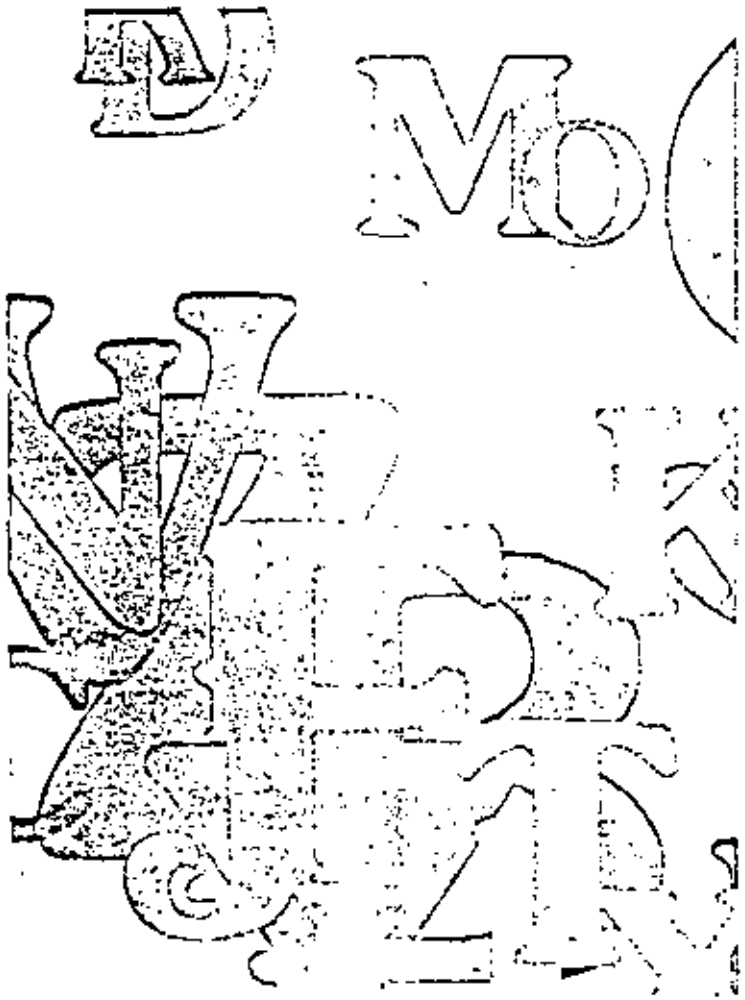
Instead of a real DO variable, you can use an integer DO variable N that varies from 0 to 100. You can compute F from N with the statement

```
F = 95.0 + N/10.0
```

This method eliminates the cumulative round-off error. Here is the complete program.

```
* TEMPERATURE CONVERSION TABLE
      REAL F, C
      INTEGER N

      DO 100, N = 0, 100
         F = 95 + N/10.0
         C = 5.0/9.0 * (F - 32)
         PRINT *, 'FAHRENHEIT: ', F, ' CELSIUS:', C
100   CONTINUE
      END
```



CHARACTER DATA

INTRODUCTION A computer is more than just a "number cruncher" that carries out mathematical computations. It is a general-purpose machine that can process symbols of all types. Many important computer applications are nonnumerical, involving very little arithmetic. For instance, consider the problem of compiling a telephone directory for a city with 200,000 phones. Sorting the customer's names in alphabetical order is an enormous job but one that involves processing not of numbers, but of character data.

In this chapter you will learn how to use character data in FORTRAN. There are **character constants**, **character variables**, and **character expressions** similar to the arithmetic expressions covered in Chapter 3. One programming consideration in dealing with such data is that each character expression has a certain length; that is, it consists of a certain number of characters.

In forming character expressions from constants and variables, you can use the **concatenation operator** to join two character strings. The reverse operation, in a sense, is extracting characters from a string by using **substring expressions**.

Using these simple tools, you can write a variety of interesting nonnumerical applications.

CHARACTER CONSTANTS

A **character** is a single letter, digit, or symbol. The collection of all the characters that a computer can represent is called the **computer's character set**. For instance, the FORTRAN character set includes the capital letters A, B, C, . . . , Z, the digits 0, 1, 2, . . . , 9, a blank space, and the symbols + − * / . , ( ) = : ' and $. Many computers also include small letters a, b, c, . . . , z, and other special symbols in their character sets.

A **character string** is a sequence of one or more characters. A character constant in FORTRAN is a character string enclosed in quotes ('). For example:

```
'ABC'
```

is a character constant representing the character string ABC. The number of characters in a character string is called its **length**. Thus 'ABC' has length 3.

A space character (sometimes called a blank, space or a blank) is a character. Thus the character constant

```
'A B'
```

consists of three characters: A, space, B. Although spaces are usually ignored in FORTRAN, they are significant within a character constant.

Any character in the machine's character set may be part of a character constant. You can even include a quote mark as a part of a character constant by representing it as two consecutive quotes:

```
'DON''T'
```

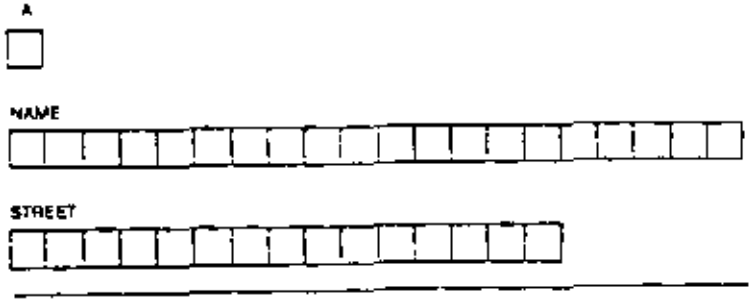
This character constant consists of five characters: D, O, N, quote T.

CHARACTER VARIABLES

Integer variables store integer values, and real variables store real values. To store a character string value, you use a character variable.

You can picture a character variable as a sequence of empty boxes in computer memory (Figure 5.1).

FIGURE 5.1



You can picture a character variable as a sequence of empty boxes in computer memory, where each box can hold one character. The type statement

```
CHARACTER A*1, NAME*20, STREET*15
```

sets up the three variables depicted here, having lengths 1, 20, and 15 characters.

Each "box," or character storage unit,* as it is more properly called, can hold one character. The number of such boxes in a character variable is called the length of the variable.

You can use a CHARACTER statement in FORTRAN to declare variables to be character variables. An example is:

```
CHARACTER NAME*20
```

This statement says two things: that NAME is a character variable, and that the length of NAME is 20 characters.

The CHARACTER statement is a type statement, like INTEGER and REAL. That means that a CHARACTER statement is nonexecutable, and must come at the beginning of your program.

You can use a CHARACTER statement to declare a single variable, as in the previous example, or to declare several character variables, as in the following statement.

```
CHARACTER A*1, NAME*20, STREET*15
```

This statement says that A, NAME, and STREET are character variables having lengths 1, 20, and 15.

The length specification that follows the asterisk after each variable name can be an integer constant, as in the previous examples, or it can be an integer expression, enclosed in parentheses, as in this statement

```
CHARACTER LINE*(2 * 40 + 1)
```

You can give a character variable any length you choose.

You can give a default length specification, after the word CHARACTER, which applies to all the variables in the CHARACTER statement. For example:

```
CHARACTER *10 X, Y, Z
```

means that X, Y, and Z are each character variables having length 10.

The default length is the length of each character variable for which you do not give an explicit length. In the statement

```
CHARACTER *10 X, Y*2, Z
```

the variables X and Z have the default length 10, but Y has length 2, since its length is declared explicitly.

*Another name for a character storage unit is a byte, meaning a group of bits (binary digits). Although character storage unit is the correct FORTRAN terminology, the word byte is in common use in computing literature.

If you provide no length specification at all, each variable has the default length of 1. Thus, the statement

```
CHARACTER A, B, C
```

means the same as

```
CHARACTER A*1, B*1, C*1
```

As with integer and real variables, you can declare character variables implicitly. For example, the statement

```
IMPLICIT CHARACTER (C, X)
```

says that unless otherwise specified, variables beginning with the letters C or X will be character variables, having the default length of 1.

You can declare character variable lengths, as well as names, implicitly. The statement

```
IMPLICIT CHARACTER*2 (A – C), CHARACTER*10 (X)
```

says that variables beginning with the letters A, B, or C are character variables of length 2, and variables beginning with the letter X are character variables of length 10.

SYMBOLIC CHARACTER CONSTANTS

In Chapter 3 you saw how to use the PARAMETER statement to define symbolic integer and real constants, as in the following example.

```
INTEGER MAXSIZ
REAL PI
PARAMETER (MAXSIZ = 100, PI = 3.1416)
```

Similarly, you can define symbolic character constants, as in the following statements.

```
CHARACTER TITLE*32
PARAMETER (TITLE = 'ACME COMPANY SALES REPORT')
```

The length of 32 was used because there are 32 characters in the constant TITLE. Another, more convenient way to set up the symbolic constant is this:

```
CHARACTER TITLE*(*)
PARAMETER (TITLE = 'ACME COMPANY SALES REPORT')
```

The difference is that the length of TITLE is given as an asterisk enclosed in

parentheses. This has the same effect as the previous definition, but it is more convenient because you need not count the characters in the constant. In general, the declaration

```
CHARACTER name*(*)
```

can be given to specify that *name* is a symbolic character constant. The actual length of *name* is then determined from the PARAMETER statement that defines its value.

EXAMPLE 5.3 Simple Grading Program

In a chemistry class, there were five laboratory assignments with 100 points possible on each. A data card has been prepared for each student, giving the student's name followed by the five laboratory grades, as in the following sample:

```
'PAULING, L.' 95, 95, 90, 87, 92
```

Notice that the name is enclosed in quotes on the data card. A student needs a total of 380 points or more to pass. A total of 450 or more is considered excellent. Write a program that prints each student's name, grades, total grade, and the word FAIL, PASS, or EXCELLENT. Some sample output is:

```
PAULING, L.    95  95  90  87  92  TOTAL  459  EXCELLENT
BOHR, N.       80  82  75  88  81  TOTAL  404  PASS
SMITH, X.      60  82   0  50  60  TOTAL  232  FAIL
```

To store the name, you can use a character variable NAME of length sufficient to store the longest name. The declaration

```
CHARACTER NAME*25
```

allows for a name of up to 25 characters. If any name is longer than 25 characters, the excess characters will just be truncated. You can read a data card with the statement

```
READ (*, *, END = 100) NAME, G1, G2, G3, G4, G5
```

There are three kinds of output line, depending on the rating of the student: FAIL, PASS, or EXCELLENT. One way to make the selection is to choose among three PRINT statements.

```
      IF (TOTAL .GE. 450) THEN
         PRINT *, NAME, G1, G2, G3, G4, G5,
     $        ' TOTAL=', TOTAL, ' EXCELLENT'
      ELSE IF (TOTAL .GE. 380) THEN
         PRINT *, NAME, G1, G2, G3, G4, G5,
     $        'TOTAL=', TOTAL, ' PASS'
      ELSE
         PRINT *, NAME, G1, G2, G3, G4, G5,
     $        ' TOTAL=', TOTAL, ' FAIL'
      END IF
```

A more concise method, however, is to use a single PRINT statement with a character variable RATING to which one of three values has been assigned. This method is illustrated in the following complete program

```
* PROGRAM TO TOTAL LAB GRADES.
*
      CHARACTER NAME*25, RATING*10
      INTEGER G1, G2, G3, G4, G5, TOTAL
10    READ (*, *, END = 100) NAME, G1, G2, G3, G4, G5
          TOTAL = G1 + G2 + G3 + G4 + G5
          IF (TOTAL .GE. 450) THEN
              RATING = 'EXCELLENT'
          ELSE IF (TOTAL .GE. 360) THEN
              RATING = 'PASS'
          ELSE
              RATING = 'FAIL'
          END IF
          PRINT *, NAME, G1, G2, G3, G4, G5
     $         ' TOTAL', TOTAL, ' ', RATING
      GO TO 10
100   END
```

CHARACTER EXPRESSIONS

Since character data is stored in memory differently than numeric data, you cannot add or subtract character data as you would numeric data. The character constant

```
'123'
```

is *not* the same as the integer constant

```
123
```

The two constants look similar on paper, but in computer memory they are entirely different. You cannot mix character with numeric data as in the *incorrect* FORTRAN statement:

```
X = '123' + 1
```

Both character constants and character variables are types of character expressions. Although the usual mathematical operations are not valid in character expressions, there is one operator you can use to combine character expressions. It is called the concatenation operator, and it is represented in FORTRAN by a double slash (//).

The word concatenation (sometimes called catenation) comes from the Greek word *catenula*, meaning *chain*. To concatenate two strings means to chain them together. If C1 and C2 are two character expressions, then

```
C1 // C2
```

is the character string obtained by joining C2 to the end of C1.

Here is an example:

```
CHARACTER F*3, C*7
F = 'FOR'
C = F // 'TRAN'
PRINT *, C
```

The second assignment statement concatenates 'FOR' and 'TRAN', assigning the value 'FORTRAN' to C.

SUBSTRINGS

As you have seen, you can think of a character variable as a sequence of boxes in computer memory. Sometimes you may want to deal with the individual boxes, or groups of boxes, that make up a character variable. That is what substrings are used for. You can think of a substring as an individual box or a group of adjacent boxes within the variable (Figure 5.2). More formally, we can define a substring of a character variable as a sequence of one or more consecutive character storage units of the variable.

Character storage units within a variable are numbered 1, 2, 3, . . . , starting with the leftmost character. To specify a substring, you need to specify the name of the

FIGURE 5.2
A SUBSTRING NAME REFERS TO PART OF A CHARACTER VARIABLE

In this figure, ALPHA is a character variable consisting of 26 character storage units. The substring name ALPHA(1:10) refers to the first 10 character storage units of ALPHA. The substring name ALPHA(13:13) refers to the 13th character storage unit. ALPHA(17:26) refers to the last 10 character storage units.

variable and the number of the first and the last character storage units. The FORTRAN notation for this is:

```
NAME(I:J)
```

This notation represents a substring name. NAME is the name of a character variable. I and J are integer expressions which are called substring expressions. The value of I is the number of the first character, and the value of J is the number of the last character in the substring. Notice that the substring expressions are separated by a colon.

The following example illustrates substrings.

```
- EXAMPLE OF SUBSTRINGS
      CHARACTER ALPHA*26, S*10, C*1
      ALPHA = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
      S = ALPHA(1:10)
      PRINT *, S
      S = ALPHA(17:26)
      PRINT *, ALPHA(1:5) // S
      PRINT *, ALPHA(1:5) // ALPHA(22:26)
      C = ALPHA(2:2)
      PRINT *, C
      END
```

The output from this program is:

```
ABCDEFGHIJ
QRSTUVWXYZ
ABCDEQRSTUVWXYZ
ABCDEVWXYZ
B
```

In this example the substring expressions are integer constants, but they could be integer expressions instead, as in the following statements.

```
N = 10
PRINT *, ALPHA(N - 5: N + 5)
```

The length of a substring name is the number of characters it contains. You can refer to the individual character storage units using substring names of length 1, like the following:

```
C(1:1)
ALPHA(N:N)
NAME(I + 1:J + 1)
```

A substring can have any length. There is only one restriction.

Rule for Substring Expressions
Whenever you use a substring name

```
NAME(I:J)
```

the values of the substring expressions I and J must obey the relations

```
1 ≤ I ≤ J ≤ length of NAME
```

Thus if you use a variable C defined by

```
CHARACTER C*10
```

it would be illegal to use substring names

```
C(9:11)
```

or

```
C(0:5)
```

because they refer to nonexistent character storage units.

A substring name always contains a colon, but you can omit either of the integer substring expressions. The notation

```
NAME(I:)
```

means the substring of NAME consisting of character number I through the end of NAME. Likewise, the notation

```
NAME(:I)
```

means the substring of NAME consisting of all the characters up to character I. For example, the statements

```
CHARACTER VOWEL*5
VOWEL = 'AEIOU'
PRINT *, VOWEL(3:)
PRINT *, VOWEL(:4)
```

will print the characters

```
IOU
AEIO
```

EXAMPLE 5.2 Shifted Output Lines
Write a program that prints the following output:

```
FORTRAN 77
 FORTRAN 77
  FORTRAN 77
   FORTRAN 77
    FORTRAN 77
          .
          .
          .
```

There are a total of 50 output lines, each line being shifted one character to the right from the previous line.

An easy way to produce the desired output is this. For each N from 1 to 50, print N blank spaces followed by the string FORTRAN 77. If BLANK is a character variable containing 50 blank spaces, you can use the substring expression

```
BLANK(:N)
```

to print the first N spaces.

```
C PRINT SHIFTED OUTPUT LINES.
C
      INTEGER N
      CHARACTER BLANK*50
      DO 10, N = 1, 50
         BLANK(N:N) = ' '
         PRINT *, BLANK(:N)//'FORTRAN 77'
10       CONTINUE
      END
```

CHARACTER ASSIGNMENT

A **character assignment statement** stores character strings, much like an arithmetic assignment statement stores numeric values. For example,

```
MONTH = 'JANUARY'
```

stores the character string JANUARY in the character variable MONTH. You have already seen several examples of the character assignment statement. This section explains its effect in more detail.

The first rule for character assignment says that you cannot mix character and numeric data in an assignment.

> *Rule for Character Assignment*
> If the left side of an assignment statement is a character variable, then the right side must be a character expression.

Thus, the following statements are illegal because 123 is an integer.

```
C ILLEGAL CHARACTER ASSIGNMENT
      CHARACTER C*3
      C = 123
```

You can use a character assignment statement to store a value in a substring. Here is an example,

```
C EXAMPLE OF STORING A VALUE
C IN A SUBSTRING.
      CHARACTER S*10
      S(1:3) = 'ABC'
      S(4:6) = 'DEF'
      S(7:) = 'GHIJ'
      PRINT *, S
      END
```

This program prints the string

```
ABCDEFGHIJ
```

In general, the rule is this:

> *Rule for Character Assignment*
> A character assignment statement has the form
>
> *name = expression*
>
> where *name* is a character variable or substring name, and *expression* is a character expression.

The variable or substring name on the left of the equal sign has a certain length, as does the expression on the right of the equal sign. If these lengths are the same, the entire value of the expression is stored. If the lengths are different, the effect is determined by the following rules.

Rules for Character Assignment
If the length of the variable is greater than the length of the expression, the value of the expression is stored in the left part of the variable and spaces are stored in the remaining characters.

If the length of the variable is less than the length of the expression, then only the leftmost characters are stored.

The following example illustrates these two rules.

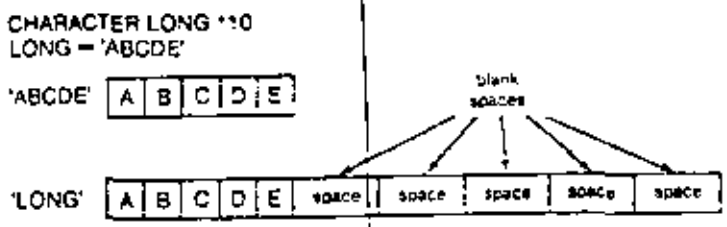
```
CHARACTER LONG *10, SHORT *3
LONG = 'ABCDE'
SHORT = 'ABCDE'
PRINT *, LONG//SHORT
END
```

The characters printed are

```
ABCDE       ABC
```

Figures 5.3 and 5.4 show the effect of the assignment statements.

FIGURE 5.3



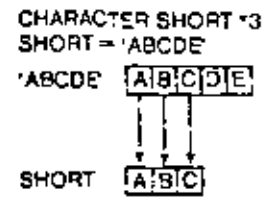
When the variable is longer than the value being stored, the value is stored in the left part of the variable and the right part is filled with blank spaces.

ANOTHER LOOK AT INPUT AND OUTPUT

Chapter 2 explained how to prepare data cards for your program. Now that you know how to write FORTRAN constants, we can give a more precise description of data card format.*

*In this section we discuss reading from a card, but the principle is the same if your program reads data from a computer terminal. Each line of terminal input corresponds to one card.

FIGURE 5.4



When the variable is shorter than the value being stored, only the leftmost characters of the value are stored.

Remember that the input list of a READ statement consists of variable names. As you now know, each variable has a certain type (integer, real, or character). The first rule for preparing input data is this:

Rule for List-Directed Input Data
Each value on the data card must be a constant of the same type as the corresponding variable in the input list.

Thus when reading an integer variable, you would supply an integer constant on the data card. When reading a real variable, you supply a real constant, which you can type in the usual form or in exponential form.

There is a convenient exception to this rule. When reading a real value, you can leave off the decimal point. To be precise:

Rule for List-Directed Input Data
When reading a value for a real variable, the corresponding data value may have the form of an integer constant.

You can read a value for a character variable or a substring name. The corresponding value on the data card must be a character constant, enclosed in quotes. The following statements read two substring names and a character variable name.

```
CHARACTER NAME *20, CLASS *10
READ *, NAME(11:20), NAME(1:10), CLASS
```

You could prepare a data card like this:

```
'JOHN' 'JONES' 'SOPHOMORE'
```

When reading character values, the length of the value is adjusted to fit the length of the variable or substring, according to the following rule.

Rule for List-Directed Input Data
If the length of a character variable or substring name in an input list is greater than the length of the corresponding data value, then the value is stored in the leftmost character positions and the remaining characters are filled with blank spaces. If the length of the variable or substring name is less than the length of the data value, only the leftmost characters are stored.

In other words, reading a value for a character variable has the same effect as storing the value in the variable with an assignment statement.

You can type one value or several values on each data card. Specifically, the rules are:

Rules for List-Directed Input Data
Each READ statement begins reading a new card. The computer will read as many cards as necessary to supply values for each new variable in the input list.

If you type more than one value on a data card, you should separate the values by a comma, or by one or more blank spaces, or by a comma and one or more blank spaces.

For example, suppose you are preparing data to be read by the following statements.

```
CHARACTER CITY *10, STATE*2
INTEGER N
REAL X
READ (*, *, END = 100) CITY, STATE, N, X
```

You could prepare data on one card like this:

```
'MIAMI', 'FL', 47, 3.5
```

Or, you could type the data without the commas, like this:

```
'MIAMI' 'FL' 47 3.5
```

Or, you could read the data from two or more cards, like this:

```
'MIAMI' 'FL'
47
3.5
```

Sometimes you may want to read several identical values from a card. Suppose you want to read five values, all zero. You could type the card like this:

```
0 0 0 0 0
```


There is a shorter way to type this input, using a repetition factor. You just type:

```
5*0
```

In input data, the asterisk does not indicate multiplication. It means to repeat the following value. So 5*0 means five values of zero. The number 5 is the repetition factor. In general, we have:

Rule for List-Directed Input Data
In input data you can write an integer number, followed by an asterisk, followed by a constant (of any type) to indicate a sequence of identical input values.

Output data format is similar to input data format. Certain details of the output format will depend on the particular computer you use. For example, the statement:

```
PRINT *, 'ABC', 1, 2.2
```

might print the three values separated by spaces, or separated by commas, or by both spaces and commas. Thus on one system the output might look like this:

```
ABC  1  2.2
```

while on another system, the output might look like this:

```
ABC,1,2.2
```

Details such as how many decimal places to print for a real number are also dependent on the particular computer system you use.

One difference between input and output data formats is that on output (to a terminal or to a line printer), there are no quotes around character values, while on input, quotes are required. The statements:

```
CHARACTER C*10
C = 'COMPUTER'
PRINT *, C
```

will print the word

```
COMPUTER
```

(with no quotes). But to read a value with the statements

```
CHARACTER C*10
READ *, C
```

you need to type quotes around the input data, like this.

```
'COMPUTER'
```

Typing quotes around character input may sometimes be inconvenient. There is a way to read character data without quotes. You can use a *format* in a READ statement. Here is an example.

```
READ (*, '(A)', END = 100) C
```

You have seen similar READ statements before. But this one, instead of having two asterisks, has the symbol

```
'(A)'
```

in place of the second asterisk. This symbol represents a format specifier consisting of an A type edit descriptor. This is often called an A format, for short.

In Chapter 8, you will learn all about formats. There are many different kinds. You can use them to specify exactly how you want your input or output data to look. For now, we will need only one kind of format, the A format illustrated above. The following rule explains how to use it.

Rule for List-Directed Input Data

Suppose that *var* is a character variable (or substring name) having a length of *n* characters. The statement

```
READ (*, '(A)', END = 900) var
```

reads the first *n* characters from a card and stores them in *var*. Any remaining characters on the card are ignored. If there are no more input cards, the program branches to statement 900 (or to whatever statement label you supply after the END =).

The following example illustrates this form of the READ statement.

EXAMPLE 5.3 Listing a Card Deck

Write a program that will read a deck of cards and list them on the printer. You could use this program to get a listing of your FORTRAN programs or data sets.

Here is a simple program that does the job. The READ statement reads all 80 characters of the card into the substring LINE(11:90). The first 10 characters of LINE are blank spaces that provide a left margin when LINE is printed.

```
      PROGRAM TO LIST A CARD DECK
      CHARACTER LINE*90
      LINE (1:10) = ' '
   10 READ (*, '(A)', END = 90) LINE(11:90)
      PRINT *, LINE
      GO TO 10
   90 END
```


EXAMPLE 5.4 Program Documentation Lines

Whenever you write a program to be used more than once, you will need to write some documentation telling how to use the program or how to change it. Often it is useful to include such documentation as comments within the program itself.

Not all of the comments in a program will be part of the general program documentation. Suppose you adopt the following convention:

1. Begin each block of general documentation with a comment card having a plus sign in column 2.
2. End each block of general documentation with a comment card having a minus sign in column 2.

Here is an example.

```
C++ PROGRAM TO LIST DOCUMENTATION.
C   AUTHOR: M. MERCHANT.
C   THIS PROGRAV READS A DECK OF INPUT CARDS AND PRINTS
C   ALL THE DOCUMENTATION BLOCKS. A DOCUMENTATION BLOCK IS
C   DEFINED TO BE ALL OF THE LINES BETWEEN A 'START' LINE AND
C   A 'STOP' LINE (INCLUDING BOTH THE START LINE AND THE STOP
C   LINE).
C   A START LINE IS A COMMENT LINE WITH A PLUS SIGN (+) IN
C   COLUMN 2.
C   A STOP LINE IS A COMMENT LINE WITH A MINUS SIGN (-) IN
C-- COLUMN 2.
C++
C   A PROGRAM MAY HAVE MORE THAN ONE DOCUMENTATION
C   BLOCK. NOTE THAT ALL LINES IN THE DOCUMENTATION BLOCK,
C-- NOT JUST COMMENTS, ARE LISTED.
```

The principle of the program is to read all the input lines, list all lines that are inside a documentation block, and skip all lines that are not. Thus, as the program reads the input, there are two possible "states:" it is either inside or outside of a documentation block. Figure 5.5 shows the top-level program design.

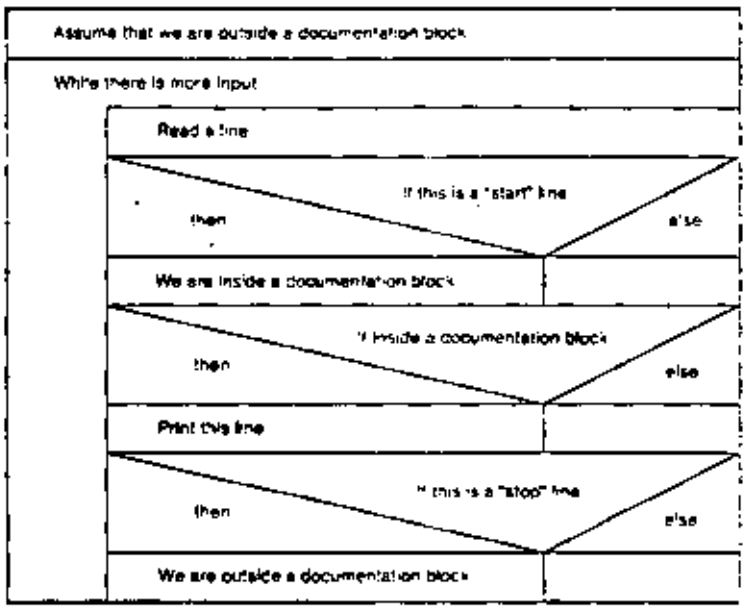
An important point to note is that the end of a documentation block is signalled by a "stop" line, but the stop line is itself part of the block. That is why the transition from being in a block to being out of a block must take place *after* the line is printed.

The program needs some way to "remember" whether it is inside a documentation block. You can use a logical variable INSIDE which is true when inside a block and false otherwise.

According to the definition, a block begins with a comment line having a plus sign in column 2. The equivalent FORTRAN condition can be written:

```
LINE(1:2) .EQ. 'C+' .OR. LINE(1:2) .EQ. 'C+'
```

FIGURE 5.5 PROGRAM DOCUMENTATION EXTRACTOR—TOP-LEVEL DESIGN



Here is the complete FORTRAN program.

```
      LOGICAL INSIDE
      CHARACTER LINE * 80

      INSIDE = .FALSE.

      READ (*, '(A)', END = 900) LINE
100       IF (LINE(1:2) .EQ. 'C+' .OR. LINE(1:2) .EQ. 'C+')
     $        INSIDE = .TRUE.

          IF (INSIDE) PRINT *, LINE

          IF (LINE(1:2) .EQ. 'C-' .OR. LINE(1:2) .EQ. 'C-')
     $        INSIDE = .FALSE.

      GO TO 100
900   END
```

COMPARING CHARACTER EXPRESSIONS

The relational operators, .LT., .EQ., and so on, can be used to compare two character expressions. With character expressions,

```
A .LT. B
```

means, especially, that A comes before B in alphabetical order. For example, the relation

```
'APPLE' .LT. 'BANANA'
```

is true, because APPLE comes before BANANA in *alphabetical order*. To see how alphabetical order works in FORTRAN, let us start with the simplest case: comparing two characters.

Alphabetic characters compare according to the usual alphabetical order. Thus the following relations are all true.

```
'A' .LT. 'B'
'B' .LT. 'C'
'A' .LT. 'Z'
```

122

What happens when you compare nonalphabetic characters? For example, how can you tell whether the relation

```
'+' .LT. '='
```

is true or false? The answer depends on the particular computer system you are using.

Suppose you took all the characters in your computer's character set and put them in their generalized alphabetical order, so that the first character is less than the second, the second character is less than the third, and so on. This arrangement is called the collating sequence of the computer's character set.

Here is one possible collating sequence for the FORTRAN character set:

```
space $ ' ( ) ++, - ./0123456789: = ABCD ... Z
```

This is the collating sequence specified by the American Standard Code for Information Interchange (ASCII), which is used on many computers.

The ASCII collating sequence, however, is only one possible choice. Most IBM computers use the Extended Binary Coded Decimal Interchange Code (EBCDIC). This is a different character set, with a different collating sequence.

Although the exact collating sequence may vary among computers, certain relations will always be true in standard FORTRAN. As we said earlier, uppercase alphabetical characters compare according to the usual alphabetical order. The characters corresponding to the digits 0 through 9 compare in the usual numerical order.

Thus, the following relations are all true.

```
'0' .LT. '1'
'1' .LT. '2'
'2' .LT. '3'
'9' .GT. '8'
```

A blank space is less than both the letter 'A' and the character '0'. Thus, the following relations are true.

```
' ' .LT. 'A'
' ' .LT. '0'
```

There are no standard rules for determining the order of other special characters. That is, the relation

```
'+' .LT. '-'
```

might be true on one computer system and false on another. Similarly, all of the following relations might be either true or false, depending on the particular collating sequence your computer uses.

```
' ' .LT. '0'
'0' .LT. 'A'
'A' .LT. '9'
'9' .LT. 'a'
```

If your computer has lowercase alphabetic characters, they will probably compare in alphabetical order, but they have no standard relation to uppercase characters. Thus,

```
'a' .LT. 'A'
```

might be either true or false, depending on your computer.

The preceding discussion deals with comparing two single characters, but you can compare two longer character expressions as well. Suppose that C1 and C2 are character expressions having the same length. Then the relation

```
C1 .EQ. C2
```

is true if and only if each character of C1 is equal to the corresponding character of C2.

To determine whether the relation

```
C1 .LT. C2
```


is true, you can use the same algorithm you use for putting two words in alphabetical order: Compare each character of C1 with the corresponding character of C2. Keep comparing until you come to a character that is different (or until you come to the end of the string). If the strings are different, then their alphabetical order is the same as the alphabetical order of the first different character. Thus the relation

```
'AAA1' .LT. 'AAA2'
```

is true because the fourth character of the first expression is less than the fourth character of the second.

You can also compare character expressions having different lengths. The effect of such a comparison is as though the shorter string were padded on the right with blank spaces to make it the same length as the longer string. Thus the relation

```
'A' // 'X' .GT. 'ABC'
```

is true.

It is illegal to compare a character expression with a numeric expression. Thus, the relation

```
'123' .EQ. 123
```

is illegal.

One final note about comparing character strings. Besides using relational operators, there are two other useful ways to do it. One way is to use the intrinsic functions LLT, LGT, LLE, and LGE, which compare strings according to the ASCII collating sequence. Another way is to use the intrinsic functions CHAR and ICHAR, which make it possible to define your own collating sequence. We will discuss intrinsic functions in Chapter 7.

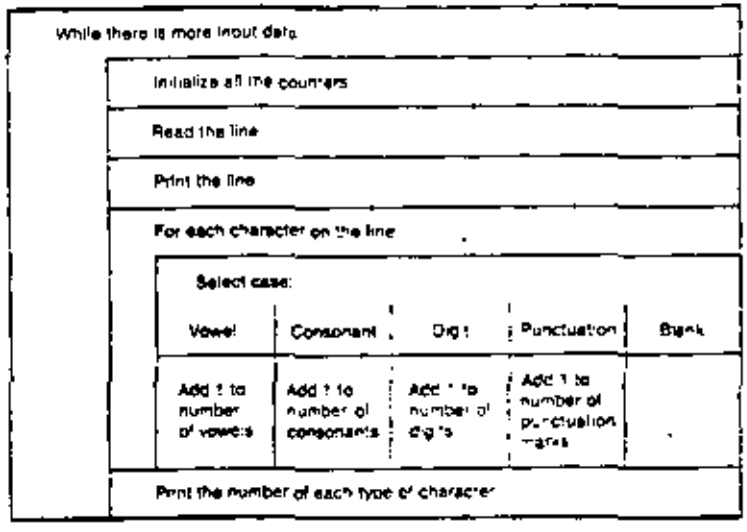
EXAMPLE 5.5 Classifying Characters

The object of this example is to read lines of text and, for each line, to produce output like the following:

```
DELAWARE RATIFIED THE U.S. CONSTITUTION ON DEC. 7, 1787.
 6 VOWELS.
22 CONSONANTS.
 5 DIGITS.
 5 PUNCTUATION MARKS.
```

The first line of output is the input line. The following four lines of output show how many characters of each type were present in the input: vowels (A, E, I, O, U), consonants (other letters of the alphabet), digits (0 through 9), and punctuation marks (which, for this example, will mean any character other than a letter, a digit, or a blank space). We assume that alphabetic letters are all uppercase.

FIGURE 5.6
PROGRAM TO CLASSIFY CHARACTERS



A top-level design for this program is shown in Figure 5.6. For each input line, you inspect each character, figure out what type it is, then add 1 to the appropriate counter.

To determine the type of a particular character C, suppose that CSET is a character variable in which the character set has been stored as follows:

```
CSET = ' AEIOUBCDFGHJKLMNPQRSTVWXYZ0123456789 '
```

You can use a while-loop structure to compare C with each character of CSET as follows:

```
      J = 1
20    IF (J .LE. 37 .AND. C .NE. CSET(J:J)) THEN
         J = J + 1
         GO TO 20
      END IF
```

The final value of J determines the type of the character C. If J = 1, C is a blank space. If $2 \le J \le 6$, C is a vowel. If $7 \le J \le 27$, C is a consonant. If $28 \le J \le 37$, C is a digit. If $J > 37$, C is not equal to any of the characters in CSET, therefore, C is a punctuation mark. Here is the complete program:

```
C
C  PROGRAM TO READ A SET OF CARDS AND PRINT THE NUMBER
C  OF VOWELS, CONSONANTS, DIGITS, AND PUNCTUATION MARKS.
C
C  VARIABLES:
C  LOOP COUNTERS:
      INTEGER I, J
C  COUNTERS FOR EACH TYPE OF CHARACTER:
      INTEGER NVOWEL, NCONS, NDIGIT, NPUNC
C  STORAGE FOR THE INPUT LINE, FOR ONE CHARACTER, AND FOR THE
C  CHARACTER SET:
      CHARACTER LINE*80, C*1, CSET*38
C
      CSET = ' AEIOUBCDFGHJKLMNPQRSTVWXYZ0123456789 '
10    READ (*, '(A)', END = 900) LINE
      PRINT *, LINE
C
C  INITIALIZE THE COUNTERS.
C
      NVOWEL = 0
      NCONS = 0
      NDIGIT = 0
      NPUNC = 0
C
C  INSPECT EACH CHARACTER OF THE LINE AND DETERMINE THE TYPE.
C
      DO 100, I = 1, 80
         C = LINE(I:I)
         J = 1
20       IF (J .LE. 37 .AND. C .NE. CSET(J:J)) THEN
            J = J + 1
            GO TO 20
         END IF
         IF (J .EQ. 1) THEN
            (IT IS A SPACE)
         ELSE IF (2 .LE. J .AND. J .LE. 6) THEN
            (IT IS A VOWEL)
            NVOWEL = NVOWEL + 1
         ELSE IF (7 .LE. J .AND. J .LE. 27) THEN
            (IT IS A CONSONANT)
            NCONS = NCONS + 1
         ELSE IF (28 .LE. J .AND. J .LE. 37) THEN
            (IT IS A DIGIT)
            NDIGIT = NDIGIT + 1
         ELSE
            (IT IS SOMETHING ELSE, THEREFORE PUNCTUATION)
            NPUNC = NPUNC + 1
         END IF
100   CONTINUE
```

```
C  OUTPUT THE RESULTS FOR THIS LINE
   PRINT *, NVOWEL, ' VOWELS,'
   PRINT *, NCONS, ' CONSONANTS,'
   PRINT *, NDIGIT, ' DIGITS,'
   PRINT *, NPUNC, ' PUNCTUATION MARKS'
   GO TO 10
900 END
```

EXAMPLE 5.6 Counting Words

The input data to this program consists of a paragraph of English prose, typed on cards. The problem is to count the number of words in the paragraph. Also count the number of nonblank characters in the paragraph and compute the average word length, in characters per word. Print the paragraph, as well as the three numbers. Here is some sample output.

```
WHAT A PIECE OF WORK IS MAN
HOW NOBLE IN REASON
HOW INFINITE IN FACULTY
IN FORM AND MOVING HOW
EXPRESS AND ADMIRABLE

23 WORDS, 84 CHARACTERS,
AN AVERAGE OF 4.08696 CHARACTERS PER WORD.
```

(The quotation is Shakespeare, *Hamlet*, Act II.)

For simplicity, assume that the text contains no members of special characters such as punctuation marks. In particular, assume that words are not hyphenated at the end of a line. Figure 5.7 shows the top-level design.

The only difficult part of this procedure is to determine when you have reached the end of a word. The problem can be simple if you adopt a simple definition of "word." We will assume that a word is a sequence of nonblank characters followed by either a blank character or by the end of the line. With this definition, you can write a most distinct and slightly modified version of the inner loop as shown in Figure 5.8.

This method almost works, but it has one flaw. In the bottom decision, what happens when you look at the last character of the line? There is no "following character," so you cannot carry out the instruction. You could deal with this problem by making the condition more complicated, to take into account whether this is the last character on the line. An easier resolution is to make sure that every line has an extra blank character at the end. Suppose the input line is 80 characters long. You can read the line into a variable LINE that is dimmed to have not 80, but 81 characters, and you redefine the last character with the statement

```
LINE(81:81) = ' '
```

You can then write the condition that controls the inner loop as follows:

> Starting with the first character in the line, look at each character except the last (the extra blank).

The bottom decision now makes sense, because there is a "following character" for every character being looked at. This technique is frequently used in processing character data. Figure 5.9 shows the revised algorithm.

FIGURE 5.7
PROGRAM TO COUNT WORDS—TOP-LEVEL DESIGN

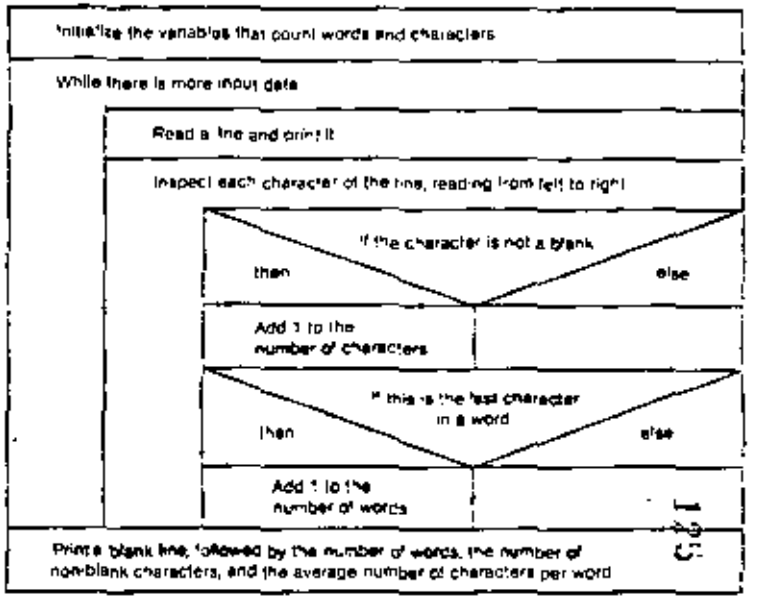


FIGURE 5.8
DETAIL OF PROCEDURE FOR COUNTING WORDS

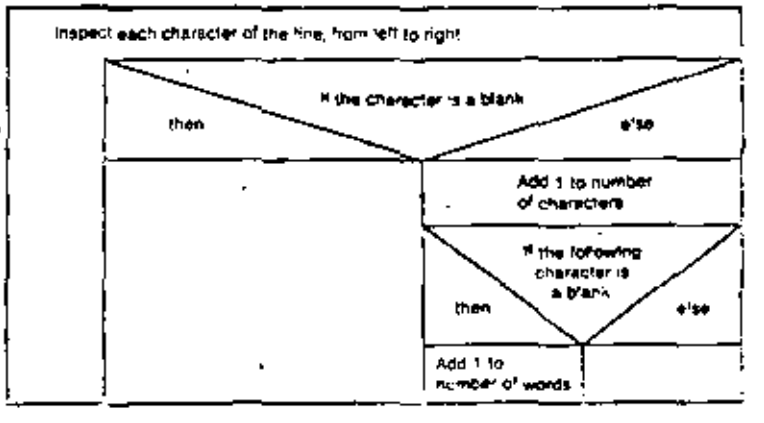
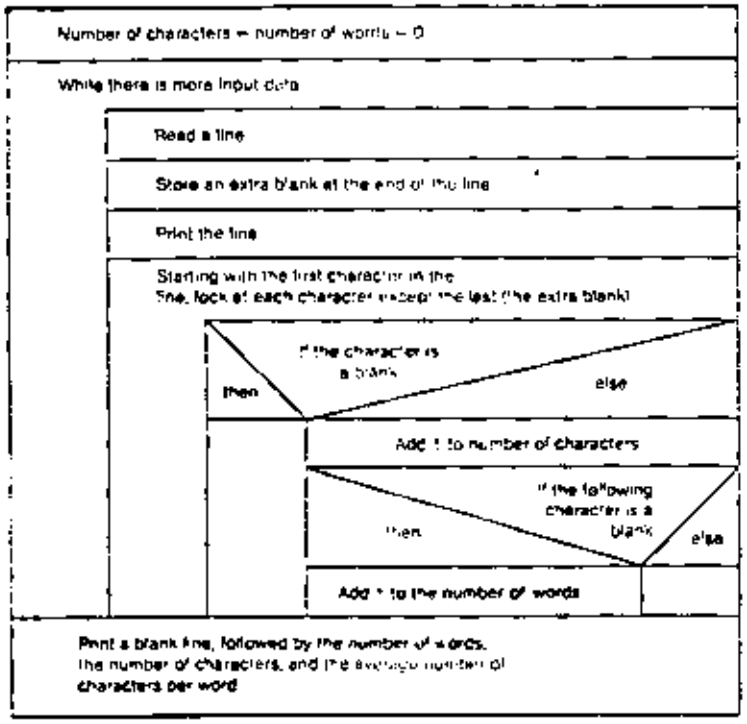


FIGURE 5.9
PROGRAM TO COUNT WORDS




```
*  PROGRAM TO COUNT WORDS AND NONBLANK CHARACTERS
*  IN A BLOCK OF TEXT, WHICH IS ASSUMED TO CONTAIN NO
*  SPECIAL CHARACTERS.
      INTEGER CHARS, I, LINSIZ, WORDS
      REAL AVER
      PARAMETER (LINSIZ = 80)
      CHARACTER BLANK, LINE*(LINSIZ + 1)
      PARAMETER (BLANK = ' ')
*
      CHARS = 0
      WORDS = 0
10    READ (*, '(A)', END = 900) LINE(1:LINSIZ)
         LINE(LINSIZ + 1:LINSIZ + 1) = BLANK
         PRINT *, LINE
         DO 20, I = 1, LINSIZ
            IF (LINE(I:I) .EQ. BLANK) THEN
               CONTINUE
            ELSE
               CHARS = CHARS + 1
               IF (LINE(I + 1:I + 1) .EQ. BLANK) THEN
                  WORDS = WORDS + 1
               END IF
            END IF
20       CONTINUE
      GO TO 10
900   PRINT *, BLANK
      AVER = CHARS
      AVER = AVER/WORDS
      PRINT *, WORDS, ' WORDS, ',
     $     CHARS, ' CHARACTERS, '
      PRINT *, ' AN AVERAGE OF ', AVER, 'CHARACTERS PER WORD.'
      END
```


SUMMARY

A character constant is a string of characters enclosed in quotes. You can also define symbolic character constants with a PARAMETER statement.

A character variable consists of a sequence of character storage units, and is declared by a CHARACTER statement. Every character variable, and, in fact, every character expression, has a certain length which is the number of character storage units it contains. You define the length of a character variable in the CHARACTER statement. The variable has the same length throughout the program.

A substring is a sequence of adjacent character storage units of a character variable. You can use a substring to extract part of a character string. The substring expressions that define the substring must be within the proper range or an execution error will result.

A character expression can be a character constant, variable, or substring name. Character expressions can be combined using the concatenation operator, which joins two strings to form a longer string.

When a value is read into a character variable or assigned to it, the length is adjusted to fit the variable. If the value is too long for the variable, the rightmost characters are truncated. If the value is shorter than the variable, blank spaces are added to the right.

When using field-directed input, each value read must be a constant of the same type as the corresponding variable. You can use an A format to read an entire input line as a single character value.

You can use relational operators to compare character expressions. For the operators .EQ. and .NE., the result of a comparison is always determined, but for the other relational operators, the result is only partially determined by the rules of standard FORTRAN. Basically, character expressions compare according to alphabetical order.

Then the variable A and X(1) are both stored in the same location, X(2) and B are stored in the same location, and so on. This is illustrated in Figure 10.3. Each variable should be of the same type as the corresponding variable.

Although the COMMON statements can be different in different program units, in practice they are frequently identical. This has the effect of making the variables in common globally defined. That is, the variables in common can be used by any program unit. This is an efficient way to pass information among subprograms, especially when much information has to be shared.

An extension of the ordinary COMMON statement is the labeled COMMON statement, which is similar, except that it allows a label for the common storage area. For example,

```
COMMON/BLOK/A, B, C
```

says to store A, B, and C in the common storage area called BLOK. The idea of this is simply to allow more than one common storage area. The general form of the labeled common statement is

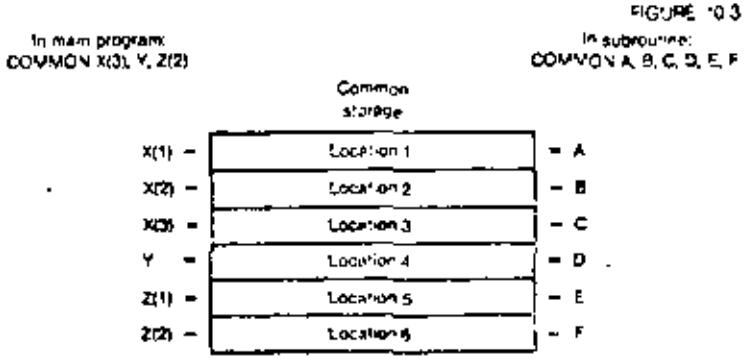
COMMON/*block*/*name*$_1$, *name*$_2$, . . . , *name*$_n$

where *block* is a one- to six-character block name, beginning with a letter, and *name*$_1$, etc., are variable or array names, just as for the unlabeled COMMON statement.

The effect of this statement is identical to that of the unlabeled common except that the variables are stored in the named common block.

Labeled common blocks with different names are separate from each other and from unlabeled common storage. This feature can be especially useful when writing a

FIGURE 10.3



Example of common statement

system of subroutines that work together. If you want to use common storage to pass information among the subroutines, you can set up a labeled common block. Then these subroutines can be used with a main program or with another system of subroutines which also uses common storage. Since the common storage blocks are labeled differently, they will not interfere with each other. There is one other difference between labeled and unlabeled common storage. Labeled common storage can be initialized with a DATA statement; unlabeled common storage cannot be.

BLOCK DATA SUBPROGRAMS

A DATA statement may not be used to initialize a variable which is in common storage. Thus the following is not allowed

```
COMMON WRONG(3)
DATA WRONG / 1., 2., 3. /
```

It is allowable, however, to use a DATA statement to initialize a variable in *labeled* common storage. This is one important difference between labeled and unlabeled common. Variables in labeled common can be initialized by DATA statements only by using a special type of subprogram called a BLOCK DATA subprogram.

The BLOCK DATA subprogram contains no executable statements. Only declaration statements (such as INTEGER, COMMON(X)) and DATA statements may be used. The BLOCK DATA subprogram begins with the statement

```
BLOCK DATA
```

or

```
BLOCK DATA name
```

where *name* is any legal subprogram name. The declarations and DATA statements follow. Like other subprograms, it is terminated by an END line.

STATEMENT FUNCTIONS

Function subprograms, as you have seen, are a very useful feature of FORTRAN. A statement function is similar to a function subprogram, but it is not a subprogram. It is a function defined by a single statement. A statement function may be defined in any program unit. Within the program unit, you can reference the statement function just like a function subprogram. Unlike a subprogram, though, a statement function cannot be referenced outside of the program unit in which it is defined.

All statement function definitions must come at the beginning of a program before the first executable statement and after all other declarative statements. The statement

function definition is a declarative statement having the same form as an assignment statement.

$$f(x_1, x_2, \ldots, x_n) = \text{expression}$$

where f is the function name, $x_1, \ldots, x_n$ are variables specifying the dummy arguments of the statement function; and *expression* is any expression which may involve the dummy arguments.

For example, the statement function definition

```
F(X) = X + SIN(X) + COS(X)
```

declares that the expression F(X) is equivalent to the expression

```
X + SIN(X) + COS(X)
```

throughout the program unit. You could replace the dummy argument X with any actual argument, just as for function subprograms. Thus,

```
PRINT *, F(A + B)
```

has the same effect as

```
PRINT *, A + B + SIN(A + B) + COS(A + B)
```

EXAMPLE 10.6 Nth Roots

You have seen examples of the SQRT function, used to compute square roots. There is no intrinsic function to compute nth roots, but you can write such a function easily as a statement function. Mathematically, the nth root of a positive number x is written

$$\sqrt[n]{x}$$

which is the same as

$$x^{\frac{1}{n}}$$

This value is computed by the statement function ROOT defined by

```
ROOT(X, N) = X**(1.0/REAL(N))
```

The following program prints a table of roots of numbers 2 through 100.

```
      ROOT TABLE
            INTEGER N
            REAL X, ROOT
            ROOT(X, N) = X**(1.0/REAL(N))
      *
            PRINT 10, (N, N = 2, 10)
      10    FORMAT('1'//T20, 'TABLE OF NTH ROOTS OF X'//
           $     T15, 'X', T20, 'N = ', T25, 9I10//)
      *
            DO 20, X = 2.0, 100.0
               PRINT 15, X, (ROOT(X, N), N = 2, 10)
      15       FORMAT (F10.0, F10.0, 9F10.7)
      20       CONTINUE
            END
```


THE SAVE STATEMENT

The **SAVE statement** is a declarative statement used to specify that certain variables in a subprogram are to remain defined even after the subprogram executes a RETURN or END statement. Normally, when a subprogram returns, all variables in the subprogram become undefined except for the following.

1. Variables in blank common storage
2. Variables that were initially defined (for example, by a DATA statement) and have not been redefined
3. Variables appearing in a named common block that appears in the subprogram and in at least one other program unit that is referencing the subprogram

The statement

```
SAVE var1, var2, ..., varn
```

means to save the values of the variables in the list when the subprogram returns. The listed variables must not be part of a labeled common block, but you can save the entire common block with a statement such as

```
SAVE block
```

where *block* is the common block name. The statement

```
SAVE
```

with no list of variables or block names means to save all variables and labeled common blocks used in the subprogram.

EXAMPLE 10.7 Random Numbers

In Chapter 7 you saw an example of a program to generate random numbers. The function RANDOM(I) worked by changing the value of its argument. The following version of the RANDOM function has no argument. Instead, it saves the value of the variable I between successive function references. The initial value of I is defined by a DATA statement.

```
      RANDOM NUMBER GENERATOR
      REAL FUNCTION RANDOM( )
      INTEGER I, J, K, V
      PARAMETER (J = 5243,
     $           K = 55397,
     $           V = 262139)
      SAVE I
      DATA I / 0 /

      I = MOD(I * J + K, V)
      RANDOM = (REAL(I) + 0.5)/REAL (V)
      END
```

The following test program will generate the same random numbers as the example in Chapter 7.

```
      INTEGER I
      REAL RANDOM
      DO 10, I = 1, 10
         PRINT *, RANDOM ( )
10       CONTINUE
      END
```

ALTERNATE RETURNS FROM A SUBROUTINE

Normally, a SUBROUTINE subprogram returns to the calling program at the statement following the CALL. There is a method by which you can have it return to some other statement in the calling program. The following example illustrates the technique:

```
      TEST OF ALTERNATE RETURN.
      DO 10, I = 1, 10
         CALL SUB(I, *5, *100)
         PRINT *, 'NORMAL RETURN'
         GO TO 10
5        PRINT *, 'ALTERNATE RETURN 1'
10       CONTINUE
      STOP
100   PRINT *, 'ALTERNATE RETURN 2'
      END
      SUBROUTINE SUB(I, *, *)
      INTEGER I
      IF (I .LE. 5) RETURN
      IF (I .LE. 9) RETURN 1
      RETURN 2
      END
```


Notice that two of the dummy arguments in SUB are asterisks. The corresponding actual arguments in the CALL statement are statement labels of statements in the main program, preceded by asterisks. In the subroutine SUB, the statement

```
RETURN 1
```

means to take the first alternate return, which means to return to the main program and GO TO statement 5. Likewise, the statement

```
RETURN 2
```

in the subprogram means to take the second alternate return, which means to return to the main program and GO TO statement 100.

In general, if a subroutine has *n* dummy arguments which are asterisks, then the statement

```
RETURN I
```

where *I* is an integer expression, means to take the *I*th alternate return. The corresponding actual argument must be an alternate return specifier of the form

```
*s
```

where *s* is a statement label of a statement in the calling program. The effect is to return to the calling and go to statement *s*. If the value of *I* is less than 1 or greater than *n*, the subroutine returns in the usual manner.

THE EXTERNAL STATEMENT

A dummy argument of a function or subroutine subprogram may itself be a subprogram name. For example, the argument F in the following function is a function itself.

```
      REAL FUNCTION DELTA(F, X)
      DELTA = F(X + 1)
      END
```

Suppose that a main program contains the statement

```
Z = DELTA(FUNC, Y)
```

where FUNC is some function subprogram name. The main program must have a way of informing the FORTRAN compiler that FUNC is a subprogram name, rather than a variable, so that the argument can be passed correctly. It does so by declaring FUNC

in an EXTERNAL statement of the form

```
EXTERNAL FUNC
```

In general, when an actual argument to a subprogram is a subprogram name, the *calling* program must declare the argument as an external reference by including an EXTERNAL statement. Note that the EXTERNAL statement goes in the program unit that does the calling, not in the program unit being called. Thus, in the example above, the EXTERNAL statement goes in the main program, not in DELTA.

The form of the EXTERNAL statement is

EXTERNAL $name_1$, $name_2$, ...

where $name_1$, $name_2$, ... are the names of subprograms to be used as arguments. This statement is a declarative statement which goes at the beginning of the program unit.

Only external subprogram names may be used as arguments. A statement function name or a generic intrinsic function name may not be an argument.

EXAMPLE 10.8 Integration by the Trapezoidal Rule

If $f(x)$ is a real-valued function, the integral $\int_a^b f(x)\,dx$ may be approximated by the *trapezoidal rule*:

$$\int_a^b f(x)\,dx = \left[(1/2)(f(a) + f(b)) + \sum_{i=1}^{n-1} f(a + i\left(\frac{b-a}{n}\right)\right]\frac{b-a}{n}$$

where n is some positive integer, giving the number of subintervals to use. The following subprogram carries out this computation.

```
* COMPUTE THE INTEGRAL OF F FROM A TO B
* USING THE TRAPEZOIDAL RULE WITH N SUBINTERVALS.
      REAL FUNCTION TRAP(F, A, B, N)
      REAL F, A, B
      INTEGER I, N
*
      TRAP = (F(A) + F(B))/2
      DO 5, I = 1, N - 1
         TRAP = TRAP + F(A + I*(B-A)/N)
5     CONTINUE
      TRAP = TRAP * (B-A)/N
      END
```

The following main program and function subprogram use the TRAP function to approximate the value of

$$\int_0^{2\pi} \sin^2(x)\,dx$$

Note that the EXTERNAL statement goes in the main program.

```
* MAIN PROGRAM TO TEST TRAP FUNCTION
      REAL SINSQ, TRAP
      EXTERNAL SINSQ
      PRINT *, TRAP(SINSQ, 0.0, 2*3.14159, 1000)
      END
* FUNCTION TO COMPUTE SIN - SQUARED OF X.
      REAL FUNCTION SINSQ(X)
      REAL X
      SINSQ = SIN(X)**2
      END
```

THE EQUIVALENCE STATEMENT

The EQUIVALENCE statement permits the use of two or more names for the same variable. It is a declaration, which comes at the beginning of the program. The form of the EQUIVALENCE statement is

EQUIVALENCE ($name_1$, $name_2$, ..., $name_n$), (...), ...

All the names enclosed in each set of parentheses refer to the same memory location throughout the program. For example,

```
EQUIVALENCE (A, B), (I, J)
```

means that throughout the program unit, A and B refer to the same memory location, and I and J both refer to another memory location. Thus,

```
      EQUIVALENCE(A, B)
      B = 0.0
      A = 1.0
      PRINT 10, B
10    FORMAT(F10.1)
```

prints the value 1.0, not 0.0. The statement A = 1.0 assigns the value 1.0 to both A and B since they are the same variable.

Array names cannot be equivalenced by this statement, but array elements can be. The declarations

```
      DIMENSION X(10)
      EQUIVALENCE(X(1), XFIRST), (X(10), XLAST)
```

let XFIRST be used as an alternate name for X(1) and XLAST as an alternate name for X(10).

To equivalence two arrays, it is sufficient to equivalence any two corresponding elements. Thus,

```
DIMENSION A(10), B(20)
EQUIVALENCE(A(5), B(1))
```

will let the array be referred to by either the name A or B. The size of the array is effectively 20, since the last ten elements of B are available to A as well.

It is allowable to equivalence variables of different types. This may cause problems, however, due to the different internal representations of numbers.

SUMMARY

A complex number in FORTRAN is a pair of real numbers which corresponds to the mathematical concept of a complex number consisting of a real part and an imaginary part. Complex constants are enclosed in parentheses. Complex variables are declared in a COMPLEX statement. Complex arithmetic in FORTRAN follows the mathematical rules for arithmetic with complex numbers. There are several intrinsic functions for calculation with complex data.

Double-precision data is like real data, but it is stored with approximately twice the precision. Thus, if a real variable is stored with seven-place precision, a double-precision value will be stored with fourteen-place precision. Double-precision constants are like E-form constants, but use the letter D in place of E. Double-precision variables are declared in a DOUBLE PRECISION statement.

The DATA statement is a declaration that stores an initial value in a variable before execution begins.

The assigned and computed GO TO statements are control statements used to branch to one of several statement labels. For the computed GO TO, the choice depends on the value of an integer expression. For the assigned GO TO, the choice is determined by a value given to a variable in an ASSIGN statement. The arithmetic IF statement is an obsolete control statement for branching to one of three statements, based on the sign of a numeric value.

Common storage provides a method of sharing data among separate program units. Blank common or labeled common storage may be declared with a COMMON statement. To initialize values in labeled common with a DATA statement, a BLOCK DATA subprogram must be used.

A statement function is defined by a one-line declaration that makes a function reference equivalent to some expression. The definition applies only to the program unit in which it appears.

The SAVE statement declares that certain variables in a subprogram are to retain their values even after the subprogram returns.

Alternate returns allow a subroutine to return to any statement in the calling program—not just to the statement following the call.

An EXTERNAL statement must be used in the calling program if some subprogram's actual argument is a subprogram name.

The EQUIVALENCE statement permits different variable names to be used as synonyms within a program unit.

VOCABULARY

alternate returns
alternate return specifier
arithmetic IF statement
ASSIGN statement
assigned GO TO statement
blank common
BLOCK DATA subprogram
COMMON statement
common storage
complex number
COMPLEX statement
computed GO TO statement
conjugate
DATA statement
double-precision number
DOUBLE PRECISION statement
EQUIVALENCE statement
EXTERNAL statement
imaginary part of a complex number
labeled common
real part of a complex number
repetition factor in a DATA statement
SAVE statement
statement function

EXERCISES

1. Let W be a complex array of four elements with

$$W_1 = 1 + i$$
$$W_2 = -1 + i$$
$$W_3 = -1 - i$$
$$W_4 = 1 - i$$

Write a program that prints a multiplication table of these complex numbers. The output will be an array that lists the products

$$W_i \cdot W_j$$

for each choice of i and j.

2. Exercise 4 of Chapter 7 (page 280) showed that a point in the plane can be represented either as a pair of coordinates (x, y), or as a pair of polar coordinates (r, θ). A point in polar coordinate form can be represented as a single complex