TUNADAN ApS
ØSTERBAKKEN 56, TUNE
4000 ROSKILDE
Tlf. 02-13 90 40

METANIC COMAL-80

USER'S MANUAL

TUNADAN ApS
ØSTERBAKKEN 56, TUNE
4000 ROSKILDE
Tlf. 02-13 90 40

Comal-80 EDP
-the key to programming

The METANIC COMAL-80 software package and documentation are copyrighted by METANIC ApS, DENMARK.

It is against the law to copy any of the software in the COMAL-80 software package on cassette tape, disk or any other medium for any purpose other than personal convenience.

It is against the law to give away or resell copies of any part of the METANIC COMAL-80 software package. Any unauthorized distribution of this product or any part thereof deprives the authors of their deserved royalties. METANIC ApS will take full legal resource against violators.

If you have any questions on these copyrights, please contact:

METANIC APS
KONGEVEJEN 177
DK-2830 VIRUM
DENMARK

ONE THING IS A SHIP TO COMMAND,
ANOTHER IS A CHART TO UNDERSTAND.

This proverb was said many years ago, long before words like byte, nanoseconds, computers, and interpreters entered our world.

Nevertheless, often during the time we worked on this manual these words came into our minds as we found it a difficult task to describe in plain words how a complicated thing like a high level language works.

However, this manual is a result of our combined efforts, and the only way we can think of the next edition being even better is by counting on you, the user, and your constructive criticism to reach the point of perfection that we desire.

Consequently, we shall be pleased to receive any correction, comment, suggestion or addition that you may have to this manual.

As the format of the manual is designed for easy updating, you may well find your contribution materialized in the next edition. For your convenience an error report is added at the end of the manual.

We have chosen to arrange all the key words in alphabetical order because an important part of the philosophy behind COMAL-80 is to make everything as easy as possible for persons not familiar with high level languages and the different groups into which the key words can be categorized.

We hope you will find working with COMAL-80 a must from now on, and that the manual will help you spend many good hours in the company of your computer.

THE AUTHORS.

ACKNOWLEDGEMENTS:

METANIC ApS hereby wishes to thank the following members of the staff and friends of COMAL-80 for their dedicated assistance in the preparation of this publication:

ROY FOX
MOGENS PELLE
ARNE CHRISTENSEN
MOGENS CHRISTENSEN
SUSANNE SONDERSTRUP

A special acknowledgement is extended to all the pioneers who helped field testing the COMAL-80 interpreter, and whose criticism and suggestions had great impact on the final specifications.

FIRST EDITION, OCTOBER 1981.
PRINTED IN DENMARK.

METANIC COMAL-80, written for the Z-80 microprocessor, is the most extensive interpreter available for microcomputers today and contains, beside a full extended BASIC, a great number of structures found in Pascal.

COMAL-80 was originally specified following specific wishes from the Danish educational field which wanted a language easy to learn, with built-in programming support and which facilitates a possible transition to other structured languages.

This manual is divided into two parts plus a number of appendices. Part 1 contains instructions for initialization of the different COMAL-80 versions and a general description of features which affect several or all the COMAL-80 instructions, while part 2 contains the syntax and semantics of all commands, statements, and functions in alphabetical order. The appendices contain the source code for the screen driver, guidelines for changing this driver for different systems, a list of error messages, demonstration programs and a list of ASCII codes.

This manual is not intended as a tutorial on the COMAL-80 language but as a reference manual for the specific features of METANIC COMAL-80.

Each of the two different COMAL-80 software packages contains two versions of the COMAL-80 interpreter. The two versions have identical features, except that the overlayed version leaves more storage to the user and uses a few seconds in the start and end of each program execution for reading the overlay file.

The different files are named:

7-digits precision:	
Non-overlayed version:	COMAL-80.COM
Overlayed version:	COMAL80S.COM
Overlay file:	COMAL-80.1
13-digits precision:	
Non-overlayed version:	COMAL80D.COM
Overlayed version:	CMAL80DS.COM
Overlay file:	COMAL80D.1

Note that each package contains the files for only one of the two possible precisions and that the CP/M operating system is not placed on the distribution floppies.

It is advised that the COMAL-80 files are copied to a new floppy, which also contains the CP/M operating system. Then remove the original disk from the computer and keep it in a safe place as this disk only, carries the warranty.

Now type the name of the version without the extension '.COM', and COMAL-80 will sign on. Note that the overlay versions will work only if the disk is placed in the CP/M default drive.

COMAL-80 being initialized the question is displayed on the terminal whether error descriptions are wanted. The user must answer this by 'Y' for yes or 'N' for no.

COMAL-80 is then ready for use which is shown by the prompt character '*' being displayed. Commands and program statements may be keyed in.

Commands are recognized by not starting with a line number, this indicates that the line is to be executed immediately following a 'RETURN'.

As commands, both the special system commands, such as 'RUN', 'LIST', etc. as well as a great deal of the COMAL-80 statements may be used enabling instant results of arithmetic and logical operations to be displayed without having to make a program.

The statement lines within COMAL-80 have the following format:

```
nnnn COMAL-80 statement [//(comment)]
```

for which nnnn is a line number within the interval of 1 to 9999. Only one statement is allowed in each line, except that more assignments may occur, separated by semicolons. For further details see the 'LET' and 'MAT' statements.

All statements may optionally be followed by a comment (also see 'REM' in chapter 2).

A COMAL-80 statement always starts with a line number, ends by 'RETURN', and may contain up to 159 characters. On terminals having a physical line length of less than this, the line, when filled, automatically continues on the following physical line.

INPUT EDITING

If an error is made as a line is being typed, move the cursor back to point at the error, and type the correct character(s). The new character(s) will replace the old one(s). The character pointed at by the cursor can be deleted by pressing the 'DEL' key (user defineable). At the same time, all characters on the right move one position left.

New characters may be inserted between already typed characters by moving the cursor back to the position where the new characters should start. Then press the 'INS' key (user defineable) and the rest of the line (including the character pointed at by the cursor) moves one position to the right leaving an empty space. This can be repeated as often as necessary to create space for any number of characters up to the maximum line length of 159 characters.

When the input is terminated by pressing the 'RETURN' key, the whole line shown on the screen is stored regardless of the cursor position.

A line, which is in the process of being typed, may be deleted by pressing the 'ESC' key (user defineable), but automatic generation of line numbers is terminated too.

To correct program lines for a program which is currently in the memory, re-type the line using the same line number or use the 'EDIT' command.

To delete the entire program currently residing in memory use the 'NEW' command.

The COMAL-80 character set comprises the alphabetic characters, numerical characters and special characters.

The alphabetic characters are the upper and lower case letters of the alphabet including { | } [\ ], which are replaced by national letters in some countries.

The numerical characters are the digits 0 through 9.

The following special characters are recognized by COMAL-80:

CHARACTER	NAME
	Blank
=	Equal sign or assignment symbol
+	Plus sign
-	Minus sign
*	Multiplication symbol
/	Slash or division symbol
^	Exponentiation symbol
(	Left parenthesis
)	Right parenthesis
#	Number sign
$	Dollar sign
!	Exclamation point
,	Comma
.	Period or decimal point
"	Double quotation marks
;	Semicolon
:	Colon
&	Ampersand
<	Less than
>	Greater than
_	Underscore
'ESC'	* Stop and wait for input
'RETURN'	Terminate input
Control-A	* Insert
Control-\	* Cursor left
Control-]	* Cursor right
Control-S	* Delete
Control-H	* Backspace
Control-U	* Cursor to start of line
Control-E	* Cursor to end of line
Control-I	* Cursor 8 step forward
Control-B	* Cursor 8 step backwards
Control-K	* Delete to end of line

* may be changed by the user.

Constants are the actual values which COMAL-80 uses during execution. There are two types of constants: string and arithmetic.

A string constant is a sequence of alphanumeric characters enclosed in double quotation marks. The length of the string is limited by the available space in the computer only.

A double quotation mark may be included in a string constant by writing 2 double quotation marks ("") immediately following each other.

Characters, which cannot be typed on the keyboard, can be included in a string constant by typing the characters' decimal ASCII codes enclosed in double quotation marks.

EXAMPLES OF STRING CONSTANTS:

```
"COMAL-80"
"$10.000"
"OPEN THAT DOOR"
"KEY ""S"" TO STOP"
"END"13""
```

Arithmetic constants are positive and negative numbers. Arithmetic constants in COMAL-80 cannot contain commas. There are two types of arithmetic constants:

1. Integer constants — Whole numbers in the range -32767 to 32767. Integer constants do not have decimal points

2. Real constants — Positive or negative real numbers, i.e. numbers that contain decimal points and positive or negative numbers represented in exponential form (similar to scientific notation). A real constant in exponential form consists of an optionally signed integer or fixed point number (the mantissa) followed by the letter 'E' and an optionally signed integer (the exponent). In addition, whole numbers outside the range for integer constants are considered real constants.

Variables are names used to represent values that are used in a COMAL-80 program. The value of a variable may be assigned explicitly by the programmer, or it may be assigned as the result of calculations in the program. Before a variable is assigned a value, it is undefined.

VARIABLE NAMES AND DECLARATION CHARACTERS

COMAL-80 variable names may be of any length up to 80 characters. The characters allowed in a variable name are letters, digits and underscore. The first character must be a letter. Special type declaration characters are also allowed. - See below.

A variable name may not be a reserved word unless the reserved word is embedded. If a variable begins with 'FN', it is assumed to be a call to a user-defined function. Reserved words include all COMAL-80 commands, statements, function names, and operator names.

Variables may represent either an arithmetic value or a string. string variable names are written with a dollar sign ($) as the last character. Integer variable names are written with a number sign (#) as the last character. The dollar sign and the number sign are variable type declaration characters, i.e. they 'declare' that the variable will represent a string or an integer.

Examples of variable names:

```
A
A8
DISKNAME$
COUNTER#
VALUE_OF_CURRENT
```

An array is a group or table of values referenced by the same variable name. Each element in an array is referenced by an array variable name that is subscripted with one arithmetic expression for each dimension. An array variable name has as many subscripts as there are dimensions in the array. When used as a parameter the array can be referenced as a whole or as an 'array of arrays' by omitting some or all the subscripts. This is described in detail in the chapter: PARAMETER SUBSTITUTION.

All arrays must be declared by a 'DIM' statement.

When an arithmetic array is declared, but before it is assigned values, all its elements have the value 0 (zero).

When a string array is declared, but before it is assigned strings, all its elements contain the string "" (string of zero length).

SUBSTRINGS.

Apart from referencing a string variable as a whole, element by element or as array of array, a part of a string variable element may be referred to.

This is done by one of the following formats:

⟨name⟩(I1,I2,...In,⟨start⟩[,⟨end⟩])
⟨name⟩(I1,I2,...In)(⟨start⟩:⟨end⟩)

In the former case, it is initially checked how many dimensions the variable ⟨name⟩ contains by means of the corresponding 'DIM' statement. If it has, say 'n' dimensons, then the first 'n' indices in the parenthesis are used to specify the actual element. Further, the parenthesis may contain one or two indices, i.e. ⟨start⟩ and ⟨end⟩. ⟨start⟩ specifies in which character position the substring starts, and ⟨end⟩ specifies in which it ends. Omitting ⟨end⟩ the substring consists of the character within the said ⟨start⟩ position only.

In the latter case, the first parenthesis contains the necessary number of indices, whereas the second parenthesis contains ⟨start⟩ and ⟨end⟩ information as described in the former case. In this case the ⟨end⟩ specification must be present and a colon is used to delimit it from the ⟨start⟩.

If ⟨name⟩ states a simple string variable the number of dimensions is considered zero and the parenthesis contain ⟨start⟩ and ⟨end⟩ only. In the latter format, the first parenthesis is omitted.

The arithmetic operators are:

Precedence	Operator	Operation	Example
1	^	Exponentiation	X^Y
2	/	Division	X/Y
2	*	Multiplication	X*Y
2	DIV	Integer division	X DIV Y
2	MOD	Modulus	X MOD Y
3	-	Negation	-X
4	+	Addition	X+Y
4	-	Subtraction	X-Y

Precedence of operators means that from an expression containing more than one, they are executed in the order decribed in the above table. More operators of the same precedence are resolved from left to right.

The precedence may be overruled by parentheses, as expressions enclosed in parentheses are resolved first. When more operators occur in the same set of parentheses the above table applies again.

Apart from negation the arithmetic operators may be used only between expressions giving arithmetic values. Negation may be used only for expressions giving arithmetic values.

The arithmetic value of a logical expression being true is 1, whereas the arithmetic value for a false logical expression is 0.

Relational operators are used to compare two values. The result of a such comparison may be either true (= 1) or false (= 0). This result may then be used to influence the program run.

Whenever an arithmetic value is used as a logical value, the number 0 is interpreted as false, and numbers different from 0 are interpreted as true.

Operator	Relation	Example
=	Equality	X=Y
⟨⟩	Inequality	X⟨⟩Y
⟩	Greater than	X⟩Y
⟨	Less than	X⟨Y
⟩=	Greater than or equal	X⟩=Y
⟨=	Less than or equal	X⟨=Y

(= is also used to assign a value to a variable.)

Relational operators are used between two expressions both giving an arithmetic value or two expressions both giving a string value.

Relational operators hold second precedence to arithmetic operators meaning that within an expression containing both types all arithmetic operators are resolved before the relational operators.

In the following example:

```
X-2⟩T+3
```

the values of 'X-2' and 'T+3' are calculated prior to the comparison of the two values.

Comparison between 2 string expressions is done character by character using the ASCII codes of each character. 'A' is less than 'E', as the ASCII code for 'A' is 65 and for 'E' it is 69.

For two strings of different lengths, the short one being equal to the beginning of the long one, the short one is the smallest. Consequently, "BLACK" is smaller than "BLACKBIRD".

Comparing two strings all characters between the double quotation marks are compared, including spaces. In this respect the aggregates "" and "number", each representing only one character when found within a string value, count as one character only, namely the character represented by the aggregate.

Filenames basically follow the CP/M naming conventions. This means that only the first 8 characters are significant and that lower case letters are converted to upper case.

Following a period, an extension of three characters may be specified. The extension can be freely chosen, except in connection with 'SAVE' and 'LOAD' commands, where the COMAL-80 system automatically provides the extension '.CSB'. It is therefore not allowed to specify any extension in these commands.

If no extension is specified, it defaults to '.CML' when the file name is used in connection with the 'ENTER' and 'LIST' commands, to '.DAT' in connection with the 'OPEN' command/statement, to '.CAT' in connection wih the 'CAT' command/statement and to '.RAN' for random files.

The whole name, including the extension is used to specify a file. This means that the two commands:

```
ENTER PROGRAM
ENTER PROGRAM.CML
```

reads the same file into memory, whereas this reads another:

```
ENTER PROGRAM.LST
```

The disk drive name is optional but is treated as an integrated part of the file name. If it is omitted, the current default disk drive is used. If it is specified, it is written in front of the file name. The disk drive name is the device name of the disk to be used (see below).

Example:

```
ENTER DK1:PROGRAM.CML
```

Note that the disk drive names do not follow the CP/M naming convention.

The disk drive name consists of the two letters 'DK' (meaning disk) and a unit number followed by a colon. Thus 'DK0:' corresponds to CP/M's 'A:', 'DK1:' corresponds to CP/M's 'B:', etc.

A similar scheme is used with the other peripheral devices, meaning that these can be used as files and thereby be the source or destination for data, according to the nature of the specific device.

The names used for the different devices are:

```
'LP:' or 'LP0:' meaning the line printer
'LP1:'          meaning the puncher
'DS:' or 'DS0:' meaning the data screen
'KB:' or 'KB0:' meaning the keyboard
```

Example:

```
10 OPEN FILE 0, "KB:", READ
20 OPEN FILE 1, "LP:", WRITE
30 DIM A$ OF 100
40 LOOP
50   INPUT FILE 0:A$
60   PRINT FILE 1:A$
70 ENDLOOP
```

When 'INIT', 'RELEASE', 'FORMAT', 'DELETE', 'GETUNIT', 'RENAME', 'UNIT', and 'CAT' are used as statements, filenames are considered string expressions and must be enclosed in double quotation marks. This is not allowed in command mode. An effect of this is that file names may be specified by any string expression, which evaluates to a legal file name.

Examples:

```
100 DELETE "DK0:PROGRAM.CML"
100 INIT "DK0:",A$
100 DELETE "DK0:"+A$+".CML"
```

COMAL-80 use its own format in disk files. The normal CP/M format can be specified by extending the filename with a '/C. Further extending the filename with a '/B' specifies the CP/M binary format.

Examples:

```
    ENTER TEST.BAK/C                    // READ CP/M ASCII FILE
100 OPEN FILE 3,"TEST.XYZ/C/B",READ //OPEN CP/M BINARY FILE
100 OPEN FILE 2,"DATA/C",WRITE      //OPEN CP/M ASCII FILE
```

One of the distinct features of COMAL-80 is the inclusion of genuine procedures with parameters.

A procedure is a named program area placed between the keywords 'PROC (name)' and 'ENDPROC (name)' and which is called by the use of the keyword 'EXEC (name)'.

They basically act like the subroutines known from BASIC, i.e. they can be called from one or several places in a program and when the procedure is finished the program execution continues in the line following the calling line. But besides this, they have other features which make them a very efficient programming tool.

Firstly, they are called by name, meaning that the programmer does not have to care about the line number in which the procedure is placed.

Secondly, the procedure is non-executable until it is called, meaning that regardsless where the procedure is placed in the program the lines inside it will be bypassed unless the procedure is actually called by an 'EXEC' statement and this call can go both forwards and backwards in the program.

Thirdly, and very important, parameters can be passed on to the procedure when it is called. This means that a procedure can react differently and operate on different data each time it is called.

There are two types of procedures, called open and closed procedures. The difference between the two is a question of how the proedure sees the variables used in the rest of the program.

The variables used in an open procedure has the same status as variables used in the main program which means that if it is assigned a new value inside the procedure, it keeps this value when the procedure is terminated and program execution resumes from the line following the calling line.

The closed procedure, however, acts in many ways like a separate program. The closed procedure has its own set of variables, which can be dimensioned and assigned values inside the procedure, but they are never able to influence the variables used outside the procedure unless some special action is taken (reference parameters and the global statement). This makes it possible to write library routines which can be used in any program without risking problems with the same variable name being used both in the procedure and in the rest of the program.

The difference between the two types of procedures can be illustrated by the following two programs:

```
1                          2

10 A:=5                    10 A:=5
20 EXEC TEST               20 EXEC TEST
30 PRINT A                 30 PRINT A
40 PROC TEST               40 PROC TEST CLOSED
50  A:=3                   50  A:=3
60  PRINT A                60  PRINT A
70 ENDPROC TEST            70 ENDPROC TEST
```

Running these 2 programs the first one will twice print the digit '3' because the assignment in line 50 will overrule the assignment in line 10. The second example will print the digits '3' AND '5' because the procedure is closed and thereby the variable in line 50 is not the same as the one in line 10 even though they have the same name. Technically speaking, the variable 'A' in example 1 is global to the procedure because the whole program can see and use it, but a variable inside a closed procedure is local and can only be used inside the procedure.

A local variable must also be assigned (line 50) or dimensioned inside the closed procedure before it is used for the first time. This means that if line 50 is deleted in the second example, the program execution will stop in line 60 with an error message telling that the variable is unknown.

Even though the separation of variable names is the basic idea behind the closed procedures, it is often convenient to make a variable name known to the main program as well as to the procedure

This can be done by the 'GLOBAL' statement as shown in the following example:

```
10 A:=3
20 EXEC TEST
30 PRINT A
40 PROC TEST CLOSED
50  GLOBAL A
60  A:=3*A
70  PRINT A
80 ENDPROC TEST
```

This program will twice print the digit '9'. Note that the 'GLOBAL' statement must be placed in the closed procedure and before the part of the procedure actually using the variable for the first time.

Closed procedures can be nested to any level that the memory allows (each level uses minimum about 50 bytes, depending on the number of variables), but the 'GLOBAL' statement only works on the level where it is actually placed. The following program will print the digit '3' (in line 100) and then stop in line 60 with an error message that the variable is unknown:

```
 10 A:=3
 20 EXEC TEST1
 30 PRINT A
 40 PROC TEST1 CLOSED
 50  EXEC TEST2
 60  PRINT A
 70 ENDPROC TEST1
 80 PROC TEST2 CLOSED
 90  GLOBAL A
100  PRINT A
110 ENDPROC TEST2
```

Another way of moving a variable into and out of a closed procedure is by means of a reference parameter. this is described in details in the chapter 'PARAMETER SUBSTITUTION'.

When a variable is dimensioned or assigned a value in a closed procedure the necessary memory is not allocated until the procedure is actually called and this memory is again de-allocated when the procedure is terminated.

Thus, no matter the number of times a procedure is called there will be no error message 'out of storage', if no such error message occurs on the first call.

This 'clearing the blackboard' also makes it possible to dimension a variable in a procedure which is called several times without conflicting with the rule that a variable cannot be re-dimensioned, and it is possible to overlay arrays and string variables used for intermediate results and thereby economize on storage by dimensioning and using these in different closed procedures.

Any procedure may call any procedure defined anywhere in the main program and it may even call itself (recursion). Note, that also recursion means nesting to a new level which uses memory and must be carefully controlled.

A closed procedure can also call an open procedure. The variables inside these two procedures will then be common for these but cannot be seen from the caller of the closed procedure.

The rules for variables in closed procedures are also applicable for the other closed structure: The user-defined function.

An important part of the COMAL-80 definition is the inclusion of procedures (and user-defined functions) with parameters, which allow decomposition of a program into smaller, named routines. These can be open (open procedures) or closed (closed procedures and user defined functions).

To move data into and out of a such routine parameters are used, i.e. list of variable names specified in the calling line (the actual parameters) and in the first line of the routine (the formal parameters). The actual parameters are then inserted in the formal parameters when the routine is called.

There are two types of parameters, namely 'call by value' and 'call by reference'.

'call by value' means that the actual value of the actual parameter is assigned to the formal parameter. This type can only move data into the routine as changes to the formal parameter do not affect the actual parameter.

'call by reference' means that the formal parameter is replaced by the actual parameter. This type can move data both into and out of a routine, and is specified by the keyword 'REF' in the formal parameter list. The above mentioned replacement happens dynamically i.e. when the routine is called and cannot be seen in program listings, which always show the formal parameters.

The following examples show the difference:

```
1                          2
10 A:=3                    10 A:=3
20 EXEC TEST(A)            20 EXEC TEST(A)
30 PRINT A                 30 PRINT A
40 PROC TEST(X)            40 PROC TEST(REF X)
50  X:=3*X                 50  X:=3*X
60  PRINT X                60  PRINT X
70 ENDPROC TEST            70 ENDPROC TEST
```

Here, in line 20 'A' is the actual parameter and 'X' in line 40 is the formal parameter.

In the first example the value '3' is assigned to 'X' when the procedure 'TEST' is called in line 20 and prints the digit '9' in line 60. After the procedure is terminated the digit '3' is printed in line 30 because the variable 'A' is in no way affected.

The other example will twice print the digit '9' because the formal parameter is replaced by the actual one and the change thereby reflected back.

Parameters are always local, meaning that changes which happen to 'call by value' parameters in a routine cannot affect a variable with the same name in the rest of the program. This is shown by the following example:

```
10 A:=3
20 B:=2
30 EXEC TEST(A)
40 PRINT A,B
50 PROC TEST(A)
60  A:=3*A
70  B:=3*B
80  PRINT A,B
90 ENDPROC TEST
```

For 'A' this program will print the digit '9' in line 80 and then the digit '3' in line 40. Both lines print the digit '6' as the value for 'B'. In other words, the formal parameter 'A' is local to the procedure and another variable than the variable used in lines 10 and 40, whereas 'B' is not a parameter (and the procedure is not closed) so it is global to the procedure, and the same variable in the whole program.

The parameter lists may contain as many parameters as the maximum line length allows (159 characters), separated by commas, but there must be the same number of parameters in both lists, and corresponding parameters must conform to type and dimension. The only exception is that an integer actual parameter can be assigned to a real formal parameter when 'call by value' is used.

Constants and expressions can be used as actual parameters when 'call by value' is used.

Example:

```
10 EXEC TEST(3*5,"ERROR")
20 PROC TEST(A,B$)
30  PRINT A
40  PRINT B$
50 ENDPROC TEST
```

Note, that a formal parameter cannot be dimensioned, as the call itself carries the necessary information.

Arrays can be used as parameters either as a whole, as an array of array or a single element, but they can only be used as reference parameters in the former two cases.

When a single element is used, the element is specified in the actual parameter list with the necessary number of indices and a variable of the same type specified in the formal parameter list.

Example:

```
 10 DIM A(3,5,2)
  .
  .
100 EXEC TEST(A(1,1,1))
  .
  .
200 PROC TEST(B)
  .
  .
300 ENDPROC TEST
```

Note, that 'B' does not need to be a referenced parameter as only a single element is used.

An array of array is used by omitting one or several of the indices from the right hand side in the actual parameter list and following the formal parameter name with a parenthesis containing the same number of commas as the number of omitted indices minus 1.

Example:

```
 10 DIM A(3,5,2)
  .
  .
100 EXEC TEST(A(1,1))
  .
  .
200 PROC TEST(REF B())
  .
  .
300 ENDPROC TEST
```

In this example one should note that the parenthesis following the formal parameter 'B' is empty because the number of omitted indices is 1.

The omitted indices are then specified when the formal parameter is used in the routine.

The following example shows this:

```
 10 DIM ARRAY_OF_VECTORS(5,3)
 20 FOR I:=1 TO 5
 30  FOR J:=1 TO 3
 40   ARRAY_OF_VECTORS(I,J):=RND(1,5)
 50  NEXT J
 60 NEXT I
 70 EXEC CHANGE_SIGN(ARRAY_OF_VECTORS(4))
 80 PROC CHANGE_SIGN(REF VECTOR()) CLOSED
 90  FOR I:=1 TO 3
100   VECTOR(I):=-VECTOR(I)
110  NEXT I
120 ENDPROC CHANGE_SIGN
130 FOR I:=1 TO 5
140  FOR J:=1 TO 3
150   PRINT ARRAY_OF_VECTORS(I,J);
160  NEXT J
170  PRINT
180 NEXT I
```

It is also possible to use a whole array as a parameter. This is done by removing all the indices in the actual parameter list and following the formal parameter with a parenthesis containing the same number of commas as the dimension of the array minus 1.

Example:

```
 10 DIM A$(5,3,2) OF 25
  .
  .
100 EXEC TEST(A$)
  .
  .
200 PROC TEST(REF B$(,,))
  .
  .
300 ENDPROC TEST
```

All of the COMAL-80 commands, statements and functions are described in this chapter. Each description is formatted as follows:

Type:	States whether it is a command, a statement or a function.
Purpose:	States for what the instruction is used.
Syntax:	Shows the correct syntax for the instruction. See below for syntax notation.
Execution:	Describes how the instruction is executed.
Example:	Shows sample programs or program segments that demonstrate the use of the instruction.
Comments:	Describes in detail how the instruction is used.

Syntax Notation.

Wherever the syntax for a statement, a command or a function is given, the following rules apply:

Items in capital letters must be input as shown, but both upper and lower case letters are usable. The latter are by COMAL-80 converted to upper case in listings.

Items in lower case letters enclosed in angle brackets (< >) are to be inserted by the user.

Items in square brackets ([ ]) are optional.

All punctuations except angle brackets and square brackets (i.e. commas, parentheses, semicolons, colons, exclamation points, slashes, number signs, plus signs, minus signs or equal signs) must be included where shown.

All reserved words must be preceded by and/or followed by a space if necessary to avoid multiple interpretations.

Type:

Arithmetic function

Purpose:

To calculate the absolute value of an arithmetic expression

Syntax:

ABS(⟨expression⟩)

Execution:

Returns the absolute value of ⟨expression⟩.

Example:

```
10 PRINT ABS(3*(-5))
```

Comments:

1. ⟨expression⟩ being arithmetic is of real or integer type
   The result will be of the same type.

Type:

Logical operator

Purpose:

To create the logical 'AND' between 2 expressions.

Syntax:

⟨expression1⟩ AND ⟨expression2⟩

Execution:

⟨expression1⟩ and ⟨expression2⟩ are evaluated and the logic 'AND' created.

Example:

```
10 INPUT A#
20 INPUT B#
30 IF A#=5 AND B#=7 THEN
40  PRINT "THE PRODUCT IS  35"
50 ELSE
60  PRINT "THE PRODUCT IS PERHAPS NOT 35"
70 ENDIF
```

Comments:

1. The operator has the truth table

⟨expression1⟩	⟨expression2⟩	result
true	true	true
true	false	false
false	true	false
false	false	false

Type:

Arithmetic function

Purpose:

Returns the arctangent of an arithmetic expression.

Syntax:

ATN(⟨expression⟩)

Execution:

Returns the arctangent of ⟨expression⟩ in radians.

Example:

```
10 INPUT A
20 PRINT ATN(A)
```

Comments:

1. ⟨expression⟩ being arithmetic is of real or integer type The result will always be real and in the interval -pi/2 to pi/2.

Type:

Command

Purpose:

To automatically generate a new line number after each 'RETURN'.

Syntax:

AUTO [⟨start⟩[,⟨step⟩]]

Execution:

Following each 'RETURN' a new line number is calculated by the latest line number used (or the value initially stated) plus the indicated step. The new number is placed in the input-buffer and displayed on the screen.
The cursor is set in position 6 ready for a new input line.

Examples:

```
AUTO
AUTO 15
AUTO 10,5
```

Comments:

1. If the ⟨start⟩ value is omitted, default 10 is used.
2. If the ⟨step⟩ value is omitted, default 10 is used.
3. If an existing line number is generated, the new line replaces the former one.
4. The automatic generation of line numbers can be interrupted at any time by pressing the 'ESC' key.
   The line in which this is done, is not stored.

Type:

String function

Purpose:

Converts an arithmetic expression to binary representation.

Syntax:

BSTR$((expression))

Execution:

(expression) being arithmetic is calculated and rounded if necessary. Then the value is converted to a binary text-string of exactly 8 characters.

Example:

```
10 DIM A$ OF 8
20 INPUT B
30 A$:=BSTR$(B)
40 PRINT A$
```

Comments:

1. (expression) being arithmetic must evaluate to a value within the closed interval 0 to 255.

Type:

Arithmetic function

Purpose:

To convert a binary number from a string to an integer value.

Syntax:

BVAL (<string expression>)

Execution:

The binary number contained in a string of exactly 8 characters is converted to integer.

Example:

```
10 DIM A$ OF 8
20 INPUT "WRITE A BINARY VALUE: ": A$
30 PRINT BVAL(A$)
```

Comments:

1. If the string contains less or more than 8 digits or if it contains anything else than binary digits, the program execution is stopped with an error message.

Type:

Statement, command

Purpose:

By use of 'CALL' assembler programs for the Z-80 microprocessor may be linked to a COMAL-80 program.

Syntax:

CALL ⟨expression⟩

Execution:

⟨expression⟩ being arithmetic is calculated and rounded if necessary. The CPU then stores all its registers and calls the specified address where the program execution is started.

Examples:

```
     CALL 256
240 CALL 53248
```

Comments:

1. For further details on the Z-80 microprocessor and its assembler codes, please refer to the manufacturers' manuals.
2. The user may use the CPU registers, however, the stack-pointer and the 8 restart addresses in page zero are used and must be re-established prior to returning to COMAL-80.
3. COMAL-80 does not utilize the interrupt facilities of the CPU. Consequently, the user may do this, also after returning to COMAL-80.
4. Return to COMAL-80 is done by terminating the assembler program using a 'RET' command.

Type:

Statement

Purpose:

The case structure is used when choosing among various program sections on the basis of an expression value.

Syntax:

```
CASE (expression) OF
WHEN (list of possibilities)
.
.
WHEN (list of possibilities)
.
.
WHEN (list of possibilities)
.
[OTHERWISE
.
.]
ENDCASE
```

Execution:

The (expression) is calculated and the 'WHEN' statements are checked one by one to find whether one of the mentioned possibilities matches the calculated value.
In the affirmative the lines from the 'WHEN' statement in question, up to the next corresponding 'WHEN', 'OTHERWISE' or 'ENDCASE' statement, are executed, after which the program continues after the 'ENDCASE' statement, provided that none of the executed lines have transferred the execution to an other part of the program.
If none of the checked values fit the value of (expression) The lines following 'OTHERWISE' will be executed.
If 'OTHERWISE' is omitted the program execution in this case stops with an error message.

Example:

```
 10 DIM A$ OF 1
 20 INPUT "PRESS THE 'A' OR THE 'B' KEY":A$
 30 CASE A$ OF
 40 WHEN "A","a"
 50  PRINT "YOU HAVE PRESSED THE 'A' KEY"
 60 WHEN "B","b"
 70  PRINT "YOU HAVE PRESSED THE 'B' KEY"
 80 OTHERWISE
 90  GOTO 20
100 ENDCASE
```

Comments:

1. The expressions contained in the 'WHEN' statements must be of the same type as ⟨expression⟩ except that integer expressions in the 'WHEN' statements are allowed if ⟨expression⟩ is of real type.
2. If more 'WHEN' statements correspond to ⟨expression⟩ only the program section corresponding to the first one is executed.

Type:

Command

Purpose:

To display the catalog of a connected background storage device.

Syntax:

```
CAT [<file name1>[,<file name2>]]
CAT <file name2>
```

Execution:

The operating system of the computer is called, stating from which device the catalog is wanted.
The contents of the catalog for the actual files are then transferred to the specified <file name2>.

Examples:

```
CAT
CAT DK1:
CAT DK1:K
CAT DK1:,DK0:ABC.DEF
CAT *.CML,LP:
CAT DK1:C???????.*,LP:
CAT LP:
```

Comments:

1. <file name2> is the name of the file to which the catalog is output.
2. <file name1> specifies partly or wholly the name(s) of the catalog entries which are to be output. A partial specification may consist of a device name only (in which case the whole catalog of that device is output), or a partial file name, where the characters '*' and '?' are used following the specification of CP/M.
3. Omitting <file name2> the catalog is displayed on the terminal.
4. Omitting <file name1> the whole catalog of the current default device is displayed.

Type:

Statement

Purpose:

To write the catalog from a background storage device into a file.

Syntax:

CAT ⟨file name⟩, FILE ⟨file No.⟩

Execution:

The operating system of the computer is called, giving the information as to which device and which file names are to be written. Then the catalog is written in ASCII format in the specified ⟨file No.⟩.

Examples:

```
100 CAT "DK1:", FILE 3
100 CAT "DK1:*.CML", FILE 2
```

Comments:

1. ⟨file name⟩ is a string expression.
2. ⟨file name⟩ specifies the files wanted from a catalog.
3. ⟨file name⟩ specifies partly or wholly the name(s) of the catalog entries which are to be output. A partial specification may consist of a device name only (in which case the whole catalog of that device is output), or a partial file name, where the characters '*' and '?' are used following the specification of CP/M.
4. ⟨file name⟩ being the empty string the whole catalog of of the current default device is displayed.
5. Before meeting the 'CAT' statement, a file carrying the stated ⟨file No.⟩ must be opened using the 'OPEN' statement.
6. The device on which the catalog is to be output is specified in the 'OPEN' statement.
7. Following a closing and a re-opening, the created file may be read by using the 'INPUT FILE' statement.
8. During programming 'FILE' and '#' are interchangeable. In program listings 'FILE' is used.

Type:

Statement

Purpose:

To load and start the execution of a program stored as a memory-image file on the background storage.

Syntax:

CHAIN ⟨file name⟩

Execution:

The memory of the computer is cleared; the program stated by ⟨file name⟩ is loaded after which the execution resumes from the lowest line number of this program.

Example:

```
10 // MAIN PROGRAM
20 DIM PROGRAM$ OF 10
30 REPEAT
40  INPUT "WHICH PROGRAM IS WANTED? ": PROGRAM$
50 UNTIL PROGRAM$="LIST" OR "UPDATE"
60 CHAIN PROGRAM$
```

Comments:

1. ⟨file name⟩ is a string expression.
2. This statement is typically used to organize a large program in smaller independent parts which are loaded and executed on the basis of user commands.
3. The program ⟨file name⟩ must be stored in a memory-image format by use of the 'SAVE' command.
4. Parameters can only be transferred to ⟨file name⟩ by means of data files.

Type:

String function

Purpose:

To convert an arithmetic expression into a single-character string.

Syntax:

CHR$(⟨expression⟩)

Execution:

⟨expression⟩ being arithmetic is calculated and rounded if necessary. The value is converted into a string consisting of a single character with that ASCII code.

Example:

```
10 INPUT A
20 PRINT CHR$(A)
```

Comments:

1. ⟨expression⟩ being arithmetic must be of a value within the closed interval of 0 to 255.

Type:

Statement, command

Purpose:

To clear the screen and place the cursor in the upper left corner.

Syntax:

CLEAR

Execution:

The screen is cleared and the cursor is placed in the upper left corner.

Examples:

```
10 CLEAR
   CLEAR
```

Comments:

1. This statement/command affects the screen only. The memory is cleared using the 'NEW' command.

Type:

Statement, command

Purpose:

To close one or more data files after use.

Syntax:

CLOSE [FILE ⟨file No.⟩]

Execution:

The data file carrying the specified ⟨file No.⟩ is closed. ⟨file No.⟩ which is an arithmetic expression is calculated and if necessary rounded prior to the closing.

Examples:

```
200 CLOSE
390 CLOSE FILE 3
540 CLOSE FILE A*B
    CLOSE
```

Comments:

1. If 'FILE' and ⟨file No.⟩ are omitted, all open data-files are closed.
2. When 'CLOSE' is executed, the stated connection between ⟨file name⟩ and ⟨file No.⟩ is detached and the file may be re-opened by the same or a new number.
3. Make sure that the 'CLOSE' statement/command is executed before the program execution is finished to avoid data being left in the system buffers.
   The 'RELEASE' command will indicate whether this is the case.
4. During programming 'FILE' and '#' are interchangeable. In program listings 'FILE' is used.

Type:

Command

Purpose:

To resume the program execution after a stop.

Syntax:

CON [<line No.>]

Execution:

The program execution is continued either in the specified <line No.> or, if a such is missing, at the point of the previous stop.

Examples:

```
CON
CON 220
```

Comments:

1. A new value may be assigned to a variable prior to resuming the program execution.
2. The program execution may be resumed after a stop created by a 'STOP' or 'END' statement, after pressing the 'ESC' key, or after a non-fatal error.
3. If the program was stopped because of an error, the program execution is resumed starting with the statement in error. In all other cases the program execution is started in the statement after the last statement executed.
4. If program editing has taken place the program execution cannot always be resumed.
5. If the program execution is interrupted by the 'ESC' key while the computer is waiting in an 'INPUT' statement, a value will not be assigned to the variable in question. In a such case the program execution should be resumed by 'CON <line No.>' for which <line No.> was displayed on the screen immediately after pressing the 'ESC' key.

Type:
Trigonometrical function.

Purpose:
To calculate the cosine of an expression.

Syntax:
COS(⟨expression⟩)

Execution:
Cosine of ⟨expression⟩, for which ⟨expression⟩ is in radians, is calculated.

Example:
```
10 INPUT A
20 PRINT COS(A)
```

Comments:
1. ⟨expression⟩ is an arithmetic expression of a real or integer type. The result will always be real.

Type:

Statement, command

Purpose:

To place the cursor in the desired position on the screen.

Syntax:

CURSOR ⟨expression1⟩, ⟨expression2⟩

Execution:

⟨expression1⟩ and ⟨expression2⟩, both of which must be arithmetic expressions, are calculated and rounded. The cursor is then moved to the character position, expressed by ⟨expression1⟩ and the line number expressed by ⟨expression2⟩.

Examples:

```
100 CURSOR 8,12
220 CURSOR CHARACTER#,LINE#
300 CURSOR 3*2,5+4
    CURSOR 10,15
```

Comments:

1. ⟨expression1⟩ is counted as positives from left to right and ⟨expression2⟩ is counted as positives from the top down. The upper left corner therefore has the coordinates 1,1.

Type:

Statement

Purpose:

To define constants in the form of a data list to be read by the 'READ' statement.

Syntax:

DATA ⟨constant1⟩,⟨constant2⟩,.....,⟨constantn⟩

Execution:

At the start of program execution, a search is made for 'DATA' statements after which they are chained into a data list. During a run, an internal pointer keeps pointing out the next constant in the list.

Example:

```
10 DIM FIRST_NAME$ OF 10
20 DIM FAMILY_NAME$ OF 15
30 DATA "JOHN","DOE"
40 READ FIRST_NAME$
50 READ FAMILY_NAME$
60 PRINT FIRST_NAME$+" "+FAMILY_NAME$
70 DATA 35
80 READ AGE
90 PRINT AGE; "YEAR"
```

Comments:

1. 'DATA' statements are non-executable and are skipped during program execution.
2. Any number of 'DATA' statements may be placed anywhere in the program.
3. A 'DATA' statement may contain as many constants (separated by commas) as allowed by the maximum length of input lines (=159 characters).
4. The 'READ' statement reads the 'DATA' statements in order of line numbers.
5. The types of constants may be mixed but must match those of the corresponding 'READ' statements. Otherwise the execution results in an error message.
   Arithmetic expressions are not allowed in a 'DATA' statement, and string constants must be enclosed in double quotation marks.
6. The constants may be re-read, partly or wholly, by means 'RESTORE', 'RESTORE ⟨line number⟩', or 'RESTORE ⟨name⟩' statements.
7. When the last constant is read the system variable 'EOD' is assigned the value of true (= 1).

Type:

Statement

Purpose:

To define and name a user-created function.

Syntax:

```
DEF FN(name)[(formal parameter list)]
.
.
ENDDEF FN(name)
```

Execution:

When finding a 'DEF' statement during a program execution, COMAL-80 skips this part of the program up to and including the corresponding 'ENDDEF' statement after which execution is resumed from the following line.
When the function is called, by the function name (if desired then followed by an actual parameter list), in an expression, the function is calculated and the value is inserted in the expression, after which the calculation is completed.

Examples:

```
10 DEF FNAB(X,Y)
20  FNAB:=X^3/Y^2
30 ENDDEF FNAB
40 I:=2
50 J:=3
60 OLE:=FNAB(I,J)
70 PRINT OLE
```

```
10 X:=2
20 Y:=3
30 DEF FNAB
40  GLOBAL X,Y
50  FNAB:=X^3/Y^2
60 ENDDEF FNAB
70 OLE:=FNAB
80 PRINT OLE
```

Comments:

1. (name) must be a legal variable name.
   (formal parameter list) is a list of the variable names of the function definition which are replaced by the actual parameter values when this function is called.
2. Variables used in a function definition are local and are used only to define the function.
   Therefore, these names may be used in other parts of the program. This independence may, however, be removed for one or more variables by a 'GLOBAL' statement.
3. Variable names in (formal parameter list) represent one by one the variable names or values as stated in the actual parameter list at the point of the call.

4. A function type may be either real or integer.
5. Only by means of global variables and the function, result values can be returned to the point of call.
6. Only simple variables (i.e. not arrays) may be used in (formal parameter list).
7. If the program section between 'DEF' and 'ENDDEF' contains statements of more lines these must all be contained in the program section.
8. The function value is returned from the function by assigning it to the function name. Otherwise the value of the function is undefined.

Type:

Command

Purpose:

To delete one or more lines from the program.

Syntax:

```
DEL ⟨start line⟩[,⟨end line⟩]
DEL ,⟨end line⟩
DEL ⟨start line⟩,
```

Execution:

The specified line(s) is/are deleted from the program.

Examples:

```
DEL 25,100
DEL ,220
DEL 95,
DEL 40
```

Comments:

1. If only ⟨start line⟩ is specified this line alone gets deleted.
2. If ⟨start line⟩ immediately followed by a comma is specified, this line and the rest of the program is deleted.
3. If a comma followed by a line number only is specified, the program is deleted up to and including this line.
4. Specifying ⟨start line⟩ comma ⟨end line⟩ the program is deleted between the former and the latter, including both.

Type:

Statement, command

Purpose:

To delete file(s) on the background storage.

Syntax:

DELETE (file name)

Execution:

The operating system is called with information on the the file(s) to be deleted.

Examples:

```
100 DELETE "TEST.CML"
220 DELETE "DK1:DATA.DAT"
300 DELETE "DK0:D???????.*"
    DELETE PROGRAM.CML
    DELETE DK1:C*.CML
```

Comments:

1. In statements (file name) is a string expression.
2. (file name) specifies partly or wholly the name(s) which is/are to be deleted where the characters '*' and/or '?' can be used following the specification of CP/M.
3. The whole file name, including any extension, must be specified.
4. In case (filename) is non-existing an error message is given for commands, but not for statements.

Type:

Statement

Purpose:

To allocate memory space for arrays and set the index limits.

Syntax:

DIM (list of indexed variables)

Execution:

Considering the type of variable the necessary memory is calculated and allocated.

Examples:

```
10 DIM MONKEY(5)
10 DIM NUMBER(7,3), COUNT(7)     // SEE NOTE 5
10 DIM CARS#(-5:15,3:8)
10 DIM A$(3:2), B(5)             // SEE NOTE 6
```

Comments:

1. Arrays must be dimensioned.
2. An array may have arbitrarily many dimensions, limited only by the memory available and the maximum length of the input line (159 characters.)
3. Each of the elements in (list of indexed variables) are specified using the syntax:
   (variable name)((list of index limits))
   where (variable name) optionally includes the declaration character '#'.
   The elements are separated using comma.
   (list of index limits) contains for each dimension the lower and upper limits for that dimension following the syntax:
   [(lower limit):](upper limit)
   The dimensions are separated by commas.
   If no lower limit is given it defaults to 1.
4. The 'DIM' statement assigns the value 0 to each element.
5. More variables can be dimensioned in the same line.
6. Arithmetic and string variables can be dimensioned on the same line.

Type:

Statement

Purpose:

To allocate memory space for strings and arrays of strings and set the index limits.

Syntax:

DIM (list of indexed variables)

Execution:

Considering the dimension and length of the variable, the necessary memory is allocated.

Examples:

```
10 DIM A$ OF 80                          // SEE NOTE 9
10 DIM A$(3) OF 10                       // SEE NOTE 7
10 DIM B$(0:1,3) OF 25                   // SEE NOTE 8
10 DIM A$(3:2) OF 10, B$(5) OF 25        // SEE NOTE 5
10 DIM A$(5) OF 15, C(5)                 // SEE NOTE 6
```

Comments:

1. Arrays and string variables must always be dimensioned.
2. An array may have arbitrarily many dimensions, limited only by the memory available and the maximum length of the input line (159 characters.)
3. Each of the elements in (list of indexed variables) are specified using the syntax:
   (variable name)[((list of index limits))] OF (length)
   where (variable name) includes the declaration character '$'.
   The elements are separated using comma.
   For arrays (list of index limits) contains for each dimension the lower and upper limits for that dimension following the syntax:
   [(lower limit):](upper limit)
   The dimensions are separated by commas.
   If no lower limit is given it defaults to 1.
   (length) indicates the maximum length of the string variable or of each of the elements in the string array. The actual value of a string variable/element may have a length varying from zero characters (the empty string) up to and including the stated (length).
4. The 'DIM' statement assigns the value "" (empty string) to each element.
5. More variables can be dimensioned in the same line.
6. Arithmetic and string variables can be dimensioned in the same line.

7. This array will contain the elements A$(1), A$(2) and A$(3) each having a maximum length of 10 characters.
8. This array will contain the elements B$(0,1), B$(0,2), B$(0,3), B$(1,1), B$(1,2) and B$(1,3) each having a maximum length of 25 characters.
9. A string variable needs not be an array.

Type:

Arithmetic operator

Purpose:

To carry out an integer division between two arithmetic expressions.

Syntax:

⟨expression1⟩ DIV ⟨expression2⟩

Execution:

⟨expression1⟩ is divided by ⟨expression2⟩ and the result is rounded to integer.

Examples:

```
100 A#:=B DIV C
100 NUMBER:=17 DIV NUM
```

Comments:

1. The result N is defined by the integer value of N which makes the expression
   ⟨expression1⟩ - N * ⟨expression2⟩
   assume its lowest possible non-negative value.
2. The calculation is carried out by executing a normal real division upon which the result is converted to integer. The type of the result depends upon the type of ⟨expression1⟩ and ⟨expression2⟩ in the following way:

⟨expression1⟩	DIV	⟨expression2⟩	result
real		real	real
real		int	real
int		real	real
int		int	int

3. Also see the 'MOD' operator.

Type:

Command

Purpose:

To make correcting easier in programs already in the computer working storage.

Syntax:

EDIT [⟨start⟩][,⟨end⟩]
EDIT [⟨start⟩,]

Execution:

The specified program area is called from the working storage and displayed on the screen line by line. The cursor is placed immediately after the last character and can be moved back and forwards on the line using the two control keys cursor left and cursor right respectively. Place the cursor on the character to be corrected, key in the correction and the cursor moves one position to the right.
Having completed the corrections, press 'RETURN' upon which the line undergoes the syntax control and when accepted it is stored. The next line is displayed and the sequence repeats until ⟨end⟩ is reached.

Examples:

EDIT
EDIT 100
EDIT 100,
EDIT ,100
EDIT 100,200

Comments:

1. If ⟨start⟩ is omitted, the editing starts at the first program line.
2. If ⟨end⟩ is omitted, the editing continues until the end of the program.
3. Omitting both limits, the editing starts in the first program line and continues until the end of the program (or until the 'ESC' key is pressed).
4. Stating only ⟨start⟩, without the comma, the editing covers this particular line only.
5. All the correction facilities described in INPUT EDITING in chapter 1 are available.

6. Also the line number may be edited which causes the line to be placed in the working storage according to the new line number. Any line already stored at that number will be deleted.
   The original line will not be deleted from the program (use the 'DEL' command).
7. When pressing 'RETURN' the line is stored in the working storage as the line is displayed on the screen regardless of the cursor position.
8. The edit command may be interrupted at any time by pressing the 'ESC' key, whereas changes in the actual line only happens when pressing 'RETURN'.

Type:

Statement

Purpose:

To stop the execution of a program

Syntax:

END

Execution:

Program execution is terminated and the prompt character '*' is displayed to show that the COMAL-80 interpreter is ready to accept new input.

Example:

```
 10 K:=0
 20 IF K>100 THEN
 30  END
 40 ELSE
 50  GOTO JOHN
 60 ENDIF
 70 LABEL JOHN
 80 PRINT K," ",
 90 K:+1
100 GOTO 20
```

Comments:

1. The 'END' statement does not give any information as to where the program execution was interrupted, as is the the case when using the 'STOP' statement.
2. The use of the 'END' statement is optional, as COMAL-80 adds a such (invisible) statement at the end of each program.
   Reaching this statement it automatically informs:

   Program execution finished

Type:

Command

Purpose:

To transfer a file from the background storage, stored as a string of ASCII characters, and place it in the working storage.

Syntax:

ENTER (file name)

Execution:

The specified file is opened and transferred character by character.
Following each 'RETURN' the line is syntax-checked and the formed line, if accepted, is placed in the working storage. In case of error the loading is temporarily halted upon which the line is displayed along with an error message.
Using the normal editing facilities the user may enter corrections, and after 'RETURN' another syntax-check takes place. When the line is accepted it is placed in the working storage after which the loading of the file continues.

Examples:

```
ENTER DKO:PROGRAM
ENTER POLYNO
```

Comments:

1. Only files stored in ASCII format, using the 'LIST' command, can be read by the 'ENTER' command.
2. The working storage is not cleared prior to the file being entered. However, new lines having a line number already existing in the working storage replace the old lines. This overriding takes place on a line-basis, with no consideration of the different lengths of lines, so that a short line can totally replace a long one. Making sure that there are no overlapping line numbers this may be used for combining two or more programs.
   In any other case, the working storage should always be cleared by using the 'NEW' command before reading a file by the 'ENTER' command.
3. ASCII files may be read by all versions of COMAL-80 why this format is recommended for storing files for a longer period of time.

Type:

System variable

Purpose:

To determine whether all data from the 'DATA' statements in the program have been read.

Syntax:

EOD

Execution:

EOD has the value of false (= 0) as long as data from the 'DATA' statements of the program are to be read. Having read the last set of data, the 'EOD' is assigned the value of true (= 1). Then executing a 'RESTORE' statement, 'EOD' again is assigned the value of false.

Example:

```
10 WHILE NOT EOD DO
20  READ A
30  PRINT A
40 ENDWHILE
50 DATA 55, 2, -15, 35
```

Type:

System variable

Purpose:

To determine whether all data in a data file have been read

Syntax:

EOF (<file No.>)

Execution:

At the execution of an 'OPEN FILE' statement or command of the type of 'READ', the corresponding 'EOF (<file No.>)' system variable is assigned the value of false (= 0). Having read the last value of the file, it is assigned the value of true (= 1).

Example:

```
10 OPEN FILE 0,"TEST",READ
20 REPEAT
30  READ FILE 0: A
40 UNTIL EOF(0)
```

Comments:

1. <file No.> is an arithmetic expression.

Type:

System variable

Purpose:

To remember whether a non-fatal error has occurred during a program execution.

Syntax:

ERR

Execution:

During a normal program execution, any error will stop the program and create an error message. However, a number of errors can be bypassed in a well-defined manner.
In such cases a program interruption may be avoided by the use of a 'TRAP ERR-' statement, before the error arises. In these cases, the system variable will be assigned a value equal to the error number, which in all tests will be considered true because it is different from 0. The program execution will then continue.

Example:

```
 10 INIT "", FILENAME$
 20 TRAP ERR-
 30 OPEN FILE 0, "XPLOCOMM", READ
 40 TRAP ERR+
 50 IF NOT ERR THEN
 60  INPUT FILE 0: DEFAULT_FILENAME$
 70 ELSE
 80  DEFAULT_FILENAME$:="XPLOPROG"
 90 ENDIF
100 CLOSE
```

Comments:

1. The execution of a program starts by assigning the value of false (= 0) to the system variable 'ERR'.
   When a 'TRAP ERR-' statement has been executed, a non-fatal error assigns its error number to 'ERR' and it retains this value until its status is checked. Immediately after a such check, 'ERR' is assigned the value of false.
   Normally, COMAL-80 sets a variable true by assigning it the value of 1, but in this case the error number is used.
   The error numbers are further described in appendix C.
2. By executing a 'TRAP ERR+' statement, the system returns to normal error handling.

Type:

String function

Purpose:

To give access to error descriptions in the COMAL-80 system

Syntax:

ERRTEXT$((expression))

Execution:

⟨expression⟩ being arithmetic is calculated and rounded if necessary. The corresponding error description is then returned.

Example:

```
10 FOR I=1 TO 295
20  PRINT ERRTEXT$(I)
30 NEXT I
```

Comments:

1. This function is only valid when error descriptions are not deleted at the start-up of COMAL-80. If they are deleted the result will be that the function returns an empty string.

Type:

System variable

Purpose:

To remember whether the 'ESC' key has been pressed.

Syntax:

ESC

Execution:

During normal program execution it is checked, before each statement, whether the 'ESC' key has been pressed. In the affirmative the program execution is stopped.
If a 'TRAP ESC-' statement has been executed, this function is blocked and the system variable 'ESC' is instead assigned the value of true (= 1) when 'ESC' is pressed.

Example:

```
10 TRAP ESC-
20 REPEAT
30  PRINT "THE 'ESC' KEY IS NOT PRESSED"
40 UNTIL ESC
50 TRAP ESC+
60 PRINT "THE 'ESC' KEY WAS PRESSED"
```

Comments:

1. Starting program execution the system variable 'ESC' is assigned the value of false (= 0). If a 'TRAP ESC-' statement is executed and the 'ESC' key pressed after that, the program execution continues but the system variable 'ESC' is assigned the value of true (= 1) and keeps this value until its status is checked.
   Immediately after the value is used, 'ESC' is again assigned the value of false (= 0).
2. The system returns to normal handling of the 'ESC' key when a 'TRAP ESC+' statement is executed.

Type:

Statement

Purpose:

To call a named sub-program and after this is finished, to return to the line following.

Syntax:

EXEC ⟨procedure name⟩[(⟨actual parameter list⟩)]

Execution:

The procedure specified by ⟨procedure name⟩ is called, as ⟨actual parameter list⟩ replaces the formal parameter list in the procedure heading.
Meeting the 'ENDPROC' statement, the program execution is resumed from the first executeable line following the 'EXEC' statement.

Examples:

```
100 EXEC TEST
100 EXEC FATAL_ERROR("ERROR IN X-PL/0-COMPILER")
100 EXEC ERROR(30)
100 EXEC ENTER_(CONSTANT#,LEV#,TX#,DX#)
100 EXEC EXPRESSION(FNINCLUDE(FSYS,RPAREN#),LEV#,TX#)
```

Comments:

1. The number of actual parameters must be the same as the number of formal parameters in the 'PROC' statement. Further, each parameter must conform to dimension and type.
2. If the formal parameter is specified by 'REF', a variable (possibly indexed) must be inserted as an actual parameter.
3. If the formal parameter is not specified by 'REF' the actual parameter must be an expression of a corresponding type, possibly just a variable name.
   Actual integer parameters may, however, be inserted in a formal real parameter.
4. The actual parameters must be defined before the 'EXEC' statement.
5. See the section 'PARAMETER SUBSTITUTION' in chapter 1 for more information.

Type:

Arithmetic function

Purpose:

Returns e to the power of an arithmetic expression.

Syntax:

EXP(⟨expression⟩)

Execution:

The base of the natural logarithm e (=2.718282) is raised to a power specified by ⟨expression⟩.

Example:

```
10 INPUT A
20 PRINT EXP(A)
```

Comments:

1. ⟨expression⟩ is an arithmetic expression of real or integer type. The result will always be real.
2. The value of ⟨expression⟩ must be less than or equal to 88.02968 by use of the COMAL-80 7-digits version and 292.4283068102 by the 13-digit version; otherwise COMAL-80 stops the program execution and creates an error message.

Type:

System constant

Purpose:

Mainly to assign a boolean variable the value of false.

Syntax:

FALSE

Execution:

Returns the value 0.

Example:

```
 10 // PRIME
 20 //
 30 DIM FLAGS#(0:8190)
 40 SIZE1:=8190
 50 //
 60 COUNT:=0
 70 MAT FLAGS#:=TRUE
 80 //
 90 FOR I:=0 TO SIZE1 DO
100  IF FLAGS#(I) THEN
110   PRIME:=I+I+3
120   K:=I+PRIME
130   WHILE K<=SIZE1 DO
140    FLAGS#(K):=FALSE
150    K:+PRIME
160   ENDWHILE
170   COUNT:+1
180  ENDIF
190 NEXT I
200 PRINT "TOTAL NUMBER OF PRIMES: ",COUNT
```

Type:

Statement

Purpose:

To delimit a program section and define the number of times it is to be executed.

Syntax:

```
FOR (variable) := (start) TO (end) [STEP (step)]
.
.
.
.
NEXT (variable)
```

Execution:

Meeting the 'FOR' statement, (variable):=(start) is assigned and it is calculated whether the inequality

((end)-(variable))*SGN ((step)))= 0

is met. If this is not the case, the 'FOR...NEXT' structure including this program section is bypassed and the execution continues from the first executable line following the 'NEXT' statement.

In case the inequality does hold, the program continues through the program section until meeting the 'NEXT' statement, then it jumps back to the line following 'FOR' adding (step) to (variable) and checks the inequality again using the new value of (variable).

This repeats until the inequality does not hold any longer.

Example:

```
10 FOR I=1 TO 100 STEP 5
20 PRINT I, " ",
30 NEXT I
40 STOP
```

Comments:

1. Omitting 'STEP (step)' the (step) value is set to 1.
2. If 'DOWNTO' is used in stead of 'TO', (step) is negated.
3. Following a 'FOR...NEXT' execution, the (variable) has the value not fulfilling the above inequality.
4. Up to 5 'FOR...NEXT' statements may be nested, each of them having their separate (variable).
   Each subroutine level is assigned a 'FOR...NEXT' depth of 5 giving the option of any depth by means of the 'GOSUB' statement or by use of procedures.

5. Each 'NEXT' statement must contain one only (variable), which must be the same one as stated in the corresponding 'FOR' statement.
6. It is possible to interrupt a 'FOR...NEXT' sequence by using 'GOTO'.
7. The start value of the (variable) is assigned before (end).
   Consequently program structures of the type:

```
10 J:= X
20 FOR J:=1 TO J+X
30 PRINT J
40 NEXT J
```

   will be executed X+1 times.
8. For each 'FOR' statement, one only 'NEXT' statement can be assigned.
9. During programming ':=' and '=' are interchangeable. In program listings ':=' is used.
10. (variable) must be an arithmetic variable.

Intentionally left blank.

Type:

Arithmetic function

Purpose:

To extract the decimal part of a real number.

Syntax:

FRAC(⟨expression⟩)

Execution:

The result is calculated according to the expression:

⟨expression⟩-INT(⟨expression⟩)

Example:

```
10 INPUT A
20 PRINT FRAC(A)
30 PRINT FRAC(5.72)
40 PRINT FRAC(-5.72)
```

Comments:

1. ⟨expression⟩ being arithmetic must be of real type. The result will be of real type.
1. ⟨expression⟩ being positive the result is calculated by cancelling the digits before the decimal point.
   If ⟨expression⟩ is negative the result is 1 minus the decimals of ⟨expression⟩.

Type:

Statement, command

Purpose:

To inform which background storage device is the present default device.

Syntax:

GETUNIT [⟨variable⟩]

Execution:

The name of the current default device is assigned to ⟨variable⟩ in the form of a 3-character code, two letters and one figure, followed by a colon.

Examples:

```
100 GETUNIT DISK$
    GETUNIT
```

Comments:

1. Using 'GETUNIT' as a command the ⟨variable⟩ must be omitted, after which the result is displayed on the terminal.
   In statements the ⟨variable⟩ must be specified.
2. The two letters indicate the type of device, for which 'DK' means floppy disk. The digit indicates the unit number.
3. ⟨variable⟩ is a string variable.

Type:

Statement

Purpose:

To make variables in the main program accessible within a 'PROC' or 'DEF' structure.

Syntax:

GLOBAL ⟨list of variable names⟩

Execution:

The variables of the main program mentioned in ⟨list of variable names⟩ are made accessible within the 'PROC' or 'DEF' structure containing the 'GLOBAL' statement.

Example:

```
10 PROC ERROR(N#) CLOSED
20  GLOBAL CC#, ERR_, ERRORS#
30  PRINT "*****"; SPC$(CC#-9); "^"; N#
40  ERR_:=FNINCLUDE(ERR_,N#+1); ERRORS#:+1
50 ENDPROC ERROR
```

Comments:

1. The variable names in ⟨list of variable names⟩ are separated by comma. Array variable names cannot be followed by any indices.
2. This statement may be used within closed procedures and 'DEF' structures only.
3. The variables are transferred from the main program even if the 'PROC' or 'DEF' structure containing the 'GLOBAL' statement is called from an other such structure.
4. The execution of the 'GLOBAL' statement does not affect the accessibility of the mentioned variables in any other part of the program than the 'PROC' or 'DEF' structure containing the 'GLOBAL' statement.
5. All operations allowed on the variables in the main program are also allowed within the 'PROC' or 'DEF' structure containing the 'GLOBAL' statement.

Type:

Statement

Purpose:

To call a subroutine, possibly from more locations in the same program, and return to the line following the call.

Syntax:

```
                    GOSUB (line number)
                    .
                    .
(line number)       .
                    .
                    RETURN
```

Execution:

Meeting a 'GOSUB' statement the program continues from the (line number) stated until meeting the 'RETURN' statement, upon which the program is resumed from the line following the calling 'GOSUB' statement.

Example:

```
10 PRINT "I START IN THE MAIN PROGRAM"
20 GOSUB 50
30 PRINT "I AM BACK IN THE MAIN PROGRAM"
40 STOP
50 PRINT "I AM IN THE SUBROUTINE"
60 RETURN
```

Comments:

1. A subroutine may be called any number of times.
2. Subroutines may be called from other subroutines, and such nestings are limited by the available memory only.
3. Following the 'RETURN' statement the program is resumed from the line immediately following the latest 'GOSUB' executed.
4. A subroutine may include more than one 'RETURN' statement.
5. Subroutines may be placed anywhere in the program, but clear identification from the main program is recommended.
6. To prevent any inadvertant execution of a subroutine it is recommended to place a 'STOP', 'GOTO', or an 'END' statement in the line immediately before the subroutine.
7. Meeting a 'RETURN' statement during an execution without having executed a 'GOSUB' statement, the program stops the execution and creates an error message.

Type:

Statement

Purpose:

To interrupt the normal sequential program execution and continue from the stated line.

Syntax:

GOTO (line number)
GOTO (name)

Execution:

The execution continues in the stated line or, if not executable, from the first executable line to follow.

Examples:

```
10 PRINT "JO",
20 GOTO 40
30 STOP
40 PRINT "HN"
50 GOTO 30
```

```
10 PRINT "JO",
20 GOTO REST
30 LABEL FINISH
40 STOP
50 LABEL REST
60 PRINT "HN"
70 GOTO FINISH
```

Comments:

1. Statements like 'LABEL' and 'REM' are among those not executable.

Type:

Statement

Purpose:

To execute or skip a statement depending on a logical expression being true or false.

Syntax:

IF ⟨logical expression⟩ [THEN] ⟨statement⟩

Execution:

Only when ⟨logical expression⟩ is true (<> 0), ⟨statement⟩ is executed.

Example:

```
10 INPUT "PRINT A NUMBER: ": A
20 IF A THEN PRINT "A <> 0"
30 IF A<0 THEN PRINT "A<0"
40 IF A=0 THEN PRINT "A=0"
50 IF A=1 THEN PRINT "A=1"
60 IF A=2 THEN PRINT "A=2"
70 IF A>2 THEN PRINT "A>2"
```

Comments:

1. Following an 'IF...THEN' statement the following statements may be used:
   CALL, CAT, CHAIN, CLEAR, CLOSE, CURSOR, DELETE, END, EXEC, EXIT, FORMAT, GETUNIT, GOSUB, GOTO, INIT, INPUT, LET, MAT, ON, OPEN, OUT, PAGE, POKE, PRINT, QUIT, RANDOM, READ, RELEASE, RENAME, RESTORE, RETURN, SELECT, STOP, TRAP, UNIT, and WRITE.
   Further, a new 'IF...THEN' statement is allowed.
2. During programming 'THEN' may be omitted as COMAL-80 automatically adds it to program listings.

Type:

Statement

Purpose:

To execute a program section if a logical expression is true; otherwise the section is skipped.

Syntax:

```
IF (logical expression) [THEN]
.
.
.
ENDIF
```

Execution:

If the (logical expression) is true (<> 0) the program section within 'IF...ENDIF' is executed. The (logical expression) being false (= 0) the program is resumed from the first executable line following the 'ENDIF' statement.

Example:

```
10 IF MEMBER#(1 OR MEMBER#)31 THEN
20  EXEC FATALERROR("ERROR IN X-PL/0-COMPILER")
30 ENDIF
```

Comments:

1. During programming 'THEN' may be omitted, as COMAL-80 automatically adds it to program listings.

Type:

Statement

Purpose:

To execute one of two program sections depending on a logical expression being true or false.

Syntax:

```
IF (logical expression) [THEN]
.
.
.
ELSE
.
.
.
ENDIF
```

Execution:

If the (logical expression) is true (<> 0) the program section surrounded by 'IF.....ELSE' is executed. The (logical expression) being false (= 0) the program section surrounded by 'ELSE...ENDIF' is executed.

Example:

```
10 INPUT "GUESS A NUMBER BETWEEN 1 AND 5": A
20 B:=RND(1,5)
30 IF A=B THEN
40  PRINT "CORRECT"
50 ELSE
60  PRINT "WRONG. THE NUMBER WAS: "; B
70 ENDIF
80 STOP
```

Comments:

1. During programming 'THEN' may be omitted as COMAL-80 automatically adds it to program listings.

Type:

Statement

Purpose:

To execute one of several program sections depending on on one of several logical expressions being true.

Syntax:

```
IF (logical expression 1) [THEN]
.
.
ELIF (logical expression 2) [THEN]
.
.
ELIF (logical expession n) [THEN]
.
.
[ELSE
.
.]
ENDIF
```

Execution:

Every (logical expression n) is checked one by one. If one is true (<> 0) the following program section is executed until meeting the corresponding 'ELIF', 'ELSE', or 'ENDIF' statement, upon which the program resumes from the first executable line following the 'ENDIF' statement.
When all (logical expressions) are false (= 0) the program section surrounded by 'ELSE...ENDIF' is executed, upon which the program is resumed from the first executable line following the 'ENDIF' statement.

Example:

```
 10 INPUT "PRESS ONE OF THE DIGITS 1, 2, OR 3: ": A,
 20 IF A=1 THEN
 30  PRINT "THE DIGIT WAS 1"
 40 ELIF A=2 THEN
 50  PRINT "THE DIGIT WAS 2"
 60 ELIF A=3 THEN
 70  PRINT "THE DIGIT WAS 3"
 80 ELSE
 90  PRINT "I ASKED FOR ONE OF THE DIGITS 1, 2, OR 3!"
100 ENDIF
```

Comments:

1. 'ELIF' is an abbreviation of 'ELSE IF'.
2. If more (logical expressions) are true, only the first one is evaluated.
3. Omitting the 'ELSE' statement, and none of the (logical expressions) are true, the program execution continues in the first line after 'ENDIF'.
4. During programming 'THEN' may be omitted, as COMAL-80 automatically adds it to program listings.

Type:

String operator

Purpose:

To check whether a text string is contained in another.

Syntax:

⟨expression1⟩ IN ⟨expression2⟩

Execution:

It is checked whether ⟨expression1⟩ is contained in ⟨expression2⟩. If it is, the logical value is true (= 1). If it is not, the logical value is false (= 0).

Example:

```
10 DIM A$ OF 15
20 DIM B$ OF 15
30 INPUT "WRITE A TEXT:     ": A$
40 INPUT "WRITE ANOTHER TEXT: B$
50 IF B$ IN A$ THEN
60  PRINT "SECOND TEXT IS PART OF FIRST TEXT"
70 ELSE
80  PRINT "SECOND TEXT IS NOT PART OF FIRST TEXT"
90 ENDIF
```

Type:

Statement, command

Purpose:

To prepare a formatted diskette, placed in the drive for use.

Syntax:

INIT [(device)]

Execution:

The stated (device) is initialized.

Examples:

```
100 INIT "DK0:"
    INIT
    INIT DK1:
```

Comments:

1. Under CP/M all disk drives are initialized and the (device) indication is not used, but if it is given, it must be the name of a disk drive. No disk files may be open when this statement/command is executed.

Type:

Machine code function

Purpose:

To read the value of one of the Z-80 microprocessor input ports.

Syntax:

INP(⟨expression⟩)

Execution:

The input port, defined by ⟨expression⟩ is read.

Example:

```
10 PRINT INP(17)
```

Comments:

1. ⟨expression⟩ must be of a value greater than or equal to 0 and less than or equal to 255.
2. ⟨expression⟩ is considered a decimal value which is rounded to integer if necessary.

Type:

Statement

Purpose:

To read and assign to variables the values received from the terminal, during program execution.

Syntax:

INPUT [⟨text⟩:] ⟨variable list⟩

Execution:

When meeting the 'INPUT' statement the program execution pauses after a possible ⟨text⟩ is displayed. As the user keys in values, they are assigned to the stated variables in ⟨variable list⟩ from left to right. Having inserted the last value the user presses 'RETURN', upon which the program execution continues.

Examples:

```
100 INPUT MONKEY, JOHN#, NAME$
100 INPUT "WRITE 3 DIGITS: ": A, B, C
```

Comments:

1. If the 'INPUT' statement contains a ⟨text⟩, this is displayed exactly as described, whereas only '?' is displayed when there is no ⟨text⟩, indicating that the computer expects some input.
2. If ⟨variable list⟩ ends by a comma the following output appears in the print-zone following. The width of the print-zones are set by using 'TAB'.
3. If ⟨variable list⟩ ends by a semicolon the following output appears immediately following the latest value presented from the keyboard.
4. More values may be entered as long as they are separated by a character which cannot be part of a numerical value such as space or comma.
5. String constants must be entered as a sequence of ASCII characters. It is only possible to insert values following a string constant if the 'RETURN' key is used to terminate each such.
   When a string constant follows an arithmetic constant COMAL-80 considers the first character, which cannot be part of the artihmetic constant, a delimiter, and starts the string constant with the next character.
6. The type of values keyed in must conform with the types stated in the 'INPUT' statement.

7. (variabe list) may contain all variable types, but arrays must be properly indexed and substrings may not be used.
8. Responding to 'INPUT' by the wrong type of value, causes the error message 'ERROR IN NUMBER' and the item must be corrected. No assignment is made until an acceptable input is given.
9. Responding to 'INPUT' with too few items, causes a '?' to be printed on the terminal and the program awaits more input.
10. Responding to 'INPUT' with too many items, causes the error message 'TOO MUCH INPUT', and the input must be corrected.

Type:

Statement

Purpose:

To read data from an ASCII data-file written by the 'PRINT (USING) FILE' statement.

Syntax:

INPUT FILE ⟨file No.⟩ [,⟨rec. No.⟩]:⟨variable list⟩

Execution:

The values of the variables in ⟨variable list⟩ are read from the file contained in ⟨file No.⟩.

Examples:

```
100 INPUT FILE 3: A$
100 INPUT FILE 0: B#, C
```

Comments:

1. Before meeting the 'INPUT FILE' statement a file must be opened and the connection established between the stated file name and the used ⟨file No.⟩ of the 'INPUT FILE' statement. This is done by the 'OPEN FILE' statement or command, and type 'READ' or 'RANDOM'.
2. The ⟨rec. No.⟩ is used only in 'RANDOM' files and is an arithmetic expression which is rounded to integer if necessary.
3. ⟨file No.⟩ is an arithmetic expression.
4. ⟨variable list⟩ may contain all variable types but arrays must be properly indexed and substrings may not be used.
5. The elements of ⟨variable list⟩ are separated by commas.
6. During programming 'FILE' and '#' are interchangeable. In program listings 'FILE' is used.
7. Comments 4, 5, and 6 to the 'INPUT' statement apply equally well here.

Type:

Arithmetic function

Purpose:

Returns the largest integer, equal to or less than a specified expression.

Syntax:

INT(⟨expression⟩)

Execution:

The largest integer less than or equal to ⟨expression⟩ is calculated.

Example:

```
10 INPUT A
20 B:=INT(A)
30 PRINT B
40 PRINT INT(5.72)
50 PRINT INT(-5.72)
```

Comments:

1. ⟨expression⟩ is of real type. The result is an integer of real type.
2. Also see the 'ROUND' and 'TRUNC' functions.

Type:

Arithmetic function

Purpose:

To convert an integer, existing as a string, to an integer of integer type.

Syntax:

IVAL(⟨string expression⟩)

Execution:

The characters in ⟨string expression⟩, which must form an integer number, are converted to integer.

Example:

```
10 DIM A$ OF 4
20 INPUT A$
30 PRINT IVAL(A$)
40 PRINT IVAL("3215")
```

Comments:

1. If the string in ⟨string expression⟩ contains other characters than digits including a possible sign, the program execution is stopped and an error message is displayed.
2. Also see the 'VAL' function.

Type:

Statement

Purpose:

To name a point in a COMAL-80 program for reference to the 'GOTO' and 'RESTORE' statements.

Syntax:

LABEL (name)

Execution:

The 'LABEL' statement is non-executable and serves only to mark a point in the program.

Example:

```
10 LABEL START
20 INPUT "WRITE A NUMBER: ": NUMBER
30 PRINT NUMBER
40 GOTO START
```

Type:

Arithmetic function.

Purpose:

Returns the actual length of a string variable.

Syntax:

LEN((variable))

Execution:

The actual number of characters in (variable) is counted.

Example:

```
10 DIM A$(1:10) OF 15
20 INPUT A$(5)
30 B#:=LEN(A$(5))
40 PRINT A$(5)
50 PRINT B#
```

Comments:

1. It is the actual contents of the (variable) that is used to determine its length. The dimensioned length is only of importance by being the maximum value of the result.

Type:

Statement

Purpose:

To assign the value of an expression to a variable.

Syntax:

[LET] ⟨variable⟩ := ⟨expression⟩

Execution:

⟨expression⟩ is calculated and the result is stored in the memory space allocated for ⟨variable⟩

Example:

```
10 LET A := 5
20 LET B := 3
30 LET SUM := A+B
40 A:+B
50 DIFFERENCE := A-B
60 PRINT SUM
70 PRINT A
80 PRINT DIFFERENCE
```

Comments:

1. The use of the word 'LET' is optional, i.e. it may be omitted as shown in line 40 of the example. In program listings 'LET' is omitted.
2. During programming '=' and ':=' are interchangeable. In program listings ':=' is used.
3. ⟨variable⟩ := ⟨variable⟩ + ⟨expression⟩ may in short be written as ⟨variable⟩ :+ ⟨expression⟩.
   ⟨variable⟩ := ⟨variable⟩ - ⟨expression⟩ may be expressed ⟨variable⟩ :- ⟨expression⟩, though the latter may not be used for string variables.
4. The type used for ⟨expression⟩ and ⟨variable⟩ must be equal, though integer values can be assigned to a real variable.
5. For string variables having ⟨expression⟩ longer than ⟨variable⟩, ⟨expression⟩ will be shortened from the right.
6. For string variables having ⟨expression⟩ shorter than ⟨variable⟩, ⟨variable⟩ gets the actual length only.
7. Assigning to substrings, ⟨expression⟩ and ⟨variable⟩ must be of the same length.
8. More assignments may be done on a single line, separated by semicolon, but the keyword 'LET' (which is optional) must only appear before the first assignment.

Type:

Command

Purpose:

To list the working storage of the computer, partly or wholly, as a string of ASCII characters.

Syntax:

```
LIST [⟨start⟩][,⟨end⟩][⟨file name⟩]
LIST [⟨start⟩,][⟨file name⟩]
```

Execution:

The specified part of of the program, being in the internal format, is converted into a string of ASCII characters and listed on the specified file.

Examples:

```
LIST
LIST 10
LIST 10,100
LIST ,100
LIST 100,
LIST TEST
LIST 10,100 TEST
LIST ,100 DK1:TEST
LIST LP0:
```

Comments:

1. Omitting ⟨file name⟩ all listings are presented on the terminal carrying the device name of 'DS0:'.
   If the specified listing contains more lines than this device is able to show in one picture, only the first page is shown and the COMAL-80 interpreter awaits that the 'SPACE BAR' is pressed to display the next page, or the 'RETURN' key for displaying the next line. Pressing the 'ESC' key will terminate the listing.
2. Omitting both ⟨start line⟩ and ⟨end line⟩ the total program is listed. Omitting only ⟨start line⟩, the listing starts at the first program line. Leaving ⟨end line⟩ out the listing continues until the end of the program. Specifying only ⟨start line⟩, without the comma, only the specified line is listed.
3. The 'LIST' command considers all listings being a transfer of characters from the memory to a file.
   Consequently, a listing on a connected printer is obtained by stating 'LP:' for a ⟨file name⟩, possibly followed by the unit number of the printer. When no unit number is specified it defaults to LP0:.

4. ⟨record size⟩ is used only for files of 'RANDOM' type and expresses the total number of bytes to be written in each record. The necessary size is calculated along the following lines:
   - Integers take 2 bytes
   - Real figures take 4 bytes at 7-digits precision, and 8 bytes at 13-digits precision.
   - Strings take 2 bytes plus one byte per character of the string.
5. Up to 8 disk files may be open at the same time. This leaves room for another 2 non-disk files to be open at the same time. If disk files are used in connection with 'SELECT OUTPUT', 'LIST', 'SAVE', 'CAT', 'ENTER', or 'LOAD', fewer than 8 disk files may be opened by 'OPEN'. A file may be open on more file numbers contemporarily, provided that the same ⟨type⟩ is used.
6. Having closed a sequential file, it cannot again accept to be written in.
7. A file type 'RANDOM' always must be re-opened using the same ⟨record size⟩ with which it was originally opened. ⟨record size⟩ can be recovered by the program:

```
10 OPEN FILE 0,"⟨filename⟩.RAN",READ
20 READ FILE 0; RECORD_SIZE#
30 PRINT RECORD_SIZE
40 CLOSE
```

Type:

Command

Purpose:

To read a binary file from the background storage.

Syntax:

LOAD <file name>

Execution:

The working storage of the computer is deleted and the operating system is called, upon which the file is read.

Examples:

```
LOAD TEST
LOAD DK1:PROGRAM
```

Comments:

1. Only binary files can be read by the 'LOAD' command, i.e. files stored by the 'SAVE' command. In catalog listings these files may be identified by the extension of the name by '.CSB'.
2. The extension '.CSB' is always supplied by the COMAL-80 system and cannot be stated by the user.

Type:

Arithmetic function

Purpose:

Returns the natural logarithm of an arithmetic expression.

Syntax:

LOG(⟨expression⟩)

Execution:

The natural logarithm of ⟨expression⟩ is calculated.

Examples:

```
10 INPUT A
20 PRINT LOG(A)
```

Comments:

1. ⟨expression⟩ is an arithmetic expression of real or integer type. The result will always be real.
2. If ⟨expression⟩ is less than or equal to 0 the program execution is stopped and followed by an error message.

Type:

Statement

Purpose:

To repeat the execution of a program section until an internal condition is fulfilled.

Syntax:

```
LOOP
.
.
.
.
ENDLOOP
```

Execution:

The program section enclosed by 'LOOP....ENDLOOP' is repeatedly executed until meeting an 'EXIT' statement in the program.
Then the program execution resumes from the first executable line following the 'ENDLOOP' statement.

Example:

```
10 NUMBER:=0
20 LOOP
30  NUMBER:+1
40  PRINT NUMBER
50  IF NUMBER=8 THEN EXIT
60 ENDLOOP
```

Comments:

1. The execution of the 'LOOP...ENDLOOP' section may also be interrupted by a 'GOTO' statement.
2. If 'LOOP...ENDLOOP' statements are nested, execution of an 'EXIT' statement will abandon execution of the innermost 'LOOP...ENDLOOP' statement containing the 'EXIT' statement only.

Type:

Statement

Purpose:

To assign values to each element in an array.

Syntax:

MAT ⟨variable⟩:=⟨expression⟩

Example:

```
10 DIM ARRAY(50)
20 MAT ARRAY:=5
```

Comments:

1. ⟨variable⟩ and ⟨expression⟩ must be of the same type. However, an integer expression may be assigned to the elements in a real array.
2. During programming '=' and ':=' are interchangeable. In program listings ':=' is used.
3. For string variables having ⟨expression⟩ longer than ⟨variable⟩, ⟨expression⟩ will be shortened from the right.
4. For string variables having ⟨expression⟩ shorter than ⟨variable⟩, ⟨variable⟩ gets the actual length only.
5. More assignments may be done on a single line, separated by semicolon, but the keyword 'MAT' must only appear before the first assignment.

Type:

Arithmetic operator

Purpose:

To return the remainder following an integer division.

Syntax:

⟨expression1⟩ MOD ⟨expression2⟩

Execution:

⟨expression1⟩ is integer divided by ⟨expression2⟩ and the remainder being ⟨expression1⟩ minus the result multiplied by ⟨expression2⟩ is found.

Example:

```
10 INPUT A
20 B:=A MOD 7
30 PRINT B
```

Comments:

1. The result N is defined by the lowest non-negative value which the expression:
   ⟨expression1⟩ - N * ⟨expression2⟩
   can assume for integer N.
2. The type of the result depends upon the type of ⟨expression1⟩ and ⟨expression2⟩ in the following way:

⟨expression1⟩ MOD	⟨expression2⟩	result
real	real	real
real	int	real
int	real	real
int	int	int

3. Also see the 'DIV' operator.

Type:

Command

Purpose:

To clear the working storage of the computer and prepare the COMAL-80 system for a new program.

Syntax:

NEW

Execution:

The internal pointers are initialized, except the system variable 'TAB'.

Example:

```
NEW
```

Comments:

1. The 'NEW' command should always be used before making a new program.
2. Also see note 2 to the 'ENTER' command.

Type:
Logic operator.

Purpose:
To negate a logic value

Syntax:
NOT ⟨expression⟩

Execution:
The logical value of ⟨expression⟩ is negated.

Example:
```
100 IF NOT ERR THEN EXEC READ_OK
```

Comments:
1. The operator has the following truth table

⟨expression⟩	result
true	false
false	true

Type:

Statement

Purpose:

From the value of an arithmetic expression to choose one line number out of many.

Syntax:

ON ⟨expression⟩ GOTO ⟨list of line numbers⟩
ON ⟨expression⟩ GOSUB ⟨list of line numbers⟩

Execution:

⟨expression⟩ is calculated and rounded to integer if necessary. Within ⟨list of line numbers⟩ the corresponding line number is chosen. ⟨expression⟩=1 corresponds to the first line number from the left; ⟨expression⟩=2 corresponds to the second line number from the left, etc.

Example:

```
10 INPUT "WRITE A NUMBER BETWEEN 1 AND 3 INCL: ": NUMBER
20 ON NUMBER GOTO 40,60,80
30 GOTO 10
40 PRINT "YOU WROTE 1"
50 GOTO FINISH
60 PRINT "YOU WROTE 2"
70 GOTO FINISH
80 PRINT "YOU WROTE 3"
90 LABEL FINISH
```

Comments:

1. Contradictive to the 'GOTO' statement, names may not be used in the 'ON...GOTO' statement.
2. If the rounded value of ⟨expression⟩ does not fulfil the inequality of:
   1 ⟨= ⟨expression⟩ ⟨= items in ⟨list of line numbers⟩
   the statement is skipped and the program is resumed from the next executable statement.
3. For 'ON...GOSUB' statements each line number in ⟨list of line numbers⟩ must be the first statement in a subroutine ended by a 'RETURN' statement.
   Meeting this, the program execution resumes in the first executable line after the 'GOSUB' statement.
   See also the 'GOSUB' statement.

Type:

Statement, command

Purpose:

To open a data file on the background storage.

Syntax:

OPEN FILE ⟨file No.⟩,⟨file name⟩,⟨type⟩[,⟨record size⟩]

Execution:

For all 'WRITE' files it is checked whether the specified ⟨file name⟩ is already on the background storage, in which case the program execution is stopped followed by an error message; otherwise the file is opened.
For 'READ' and 'RANDOM' files it is checked whether the ⟨file name⟩ is already on the back-up storage.
If not so, 'READ' gives an error message, whereas at 'RANDOM' the file is created. Then ⟨file name⟩ and ⟨file number⟩ are coupled so that all references to ⟨file name⟩ is done by ⟨file number⟩ until the file is closed by a 'CLOSE' statement or command.

Examples:

```
100 OPEN FILE 2,"TEST",WRITE
100 OPEN FILE 0,"DK1:DATA.RAN",RANDOM,40
```

Comments:

1. ⟨file number⟩ is an arithmetic expression which must meet one of the following values 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9, after a possible rounding.
2. ⟨file name⟩ is a string expression. Please note that not all operating systems allow that many characters in file names. For example, CP/M allows only 8 characters, being the reason why only 8 characters are transferred to the diskette.
3. ⟨type⟩ specifies how the file is used. Following possibilities are at hand:

READ	Reads sequentially from the file
WRITE	Writes sequentially in the file
RANDOM	Reads and writes the file

4. <block size> is used only for files of 'RANDOM' type and expresses the total number of bytes to be written in each record. The necessary size is calculated along the following lines:
   - Integers take 2 bytes
   - Real figures take 4 bytes at 7-digits precision, and 8 bytes at 13-digits precision.
   - Strings take 2 bytes plus as many bytes as the dimensioned maximum number of characters in the string.
5. Up to 8 disk files may be open at the same time. This leaves room for another 2 non-disk files to be open at the same time. If disk files are used in connection with 'SELECT OUTPUT', 'LIST', 'SAVE', 'CAT', 'ENTER', or 'LOAD', fewer than 8 disk files may be opened by 'OPEN'. A file may be open on more file numbers contemporarily, provided that the same <type> is used.
6. Having closed a sequential file, it cannot again accept to be written in.
7. A file type 'RANDOM' always must be re-opened using the same <record size> with which it was originally opened.

Type:

Logical operator.

Purpose:

Returns the logic 'OR' between two expressions.

Syntax:

⟨expression1⟩ OR ⟨expression2⟩

Execution:

⟨expression1⟩ and ⟨expression2⟩ are evaluated and if equal to zero considered false, else true. The logic 'OR' is then created.

Example:

```
100 IF END_DATA1 OR END_DATA2 THEN EXEC END_DATA
```

Comments:

1. The operator has the following truth table:

⟨expression1⟩	⟨expression2⟩	result
true	true	true
true	false	true
false	true	true
false	false	false

Type:

Arithmetic function

Purpose:

To convert the first character in a string into its ASCII number.

Syntax:

ORD(<string expression>)

Execution:

Returns the ASCII value of the first character in <string expression>.

Example:

```
10 DIM A$ OF 1
20 INPUT A$
30 PRINT ORD(A$)
```

Comments:

1. The result is an integer and will be greater than or equal to 0 and less than or equal to 255.

Type:

Machine language function

Purpose:

To send a byte to a machine output port.

Syntax:

OUT⟨expression1⟩,⟨expression2⟩

Execution:

The value of ⟨expression1⟩ and ⟨expression2⟩ are evaluated and rounded if necessary. The value of ⟨expression2⟩ is send to the machine output port corresponding to ⟨expression1⟩.

Example:

```
10 INPUT A
20 OUT 15,A
```

Comments:

1. The value of ⟨expression1⟩ and ⟨expression2⟩ must be a real or integer number greater than or equal to 0 and less than or equal to 255.
2. Also see 'INP'.

Type:

Statement, command

Purpose:

To advance the paper on a connected line printer to the top of the next page.

Syntax:

PAGE

Execution:

The line feed character (0AH) is transmitted to the line printer until reaching the top of the next page.

Examples:

```
100 PAGE
    PAGE
```

Comments:

1. Page shift is controlled by a counter within COMAL-80. Therefore, it is important that the paper is inserted correctly in the printer, and is not fed manually.
2. This statement/command only works for the printer with the device name 'LP0:' (or 'LP:').

Type:

Machine language function

Purpose:

To determine the value of a memory position determined by an arithmetic expression.

Syntax:

PEEK(⟨expression⟩)

Execution:

The value of ⟨expression⟩ is evaluated and rounded if necessary. The value of the corresponding memory address is returned.

Example:

```
 10 DIM B$ OF 1
 20 TRAP ESC-
 30 EXEC GET_CHR_ESC(B$)
 40 PRINT B$
 50 PROC GET_CHR_ESC(REF A$)
 60  // GET KEYBOARD INPUT WITHOUT ECHO ON THE SCREEN
 70  // THE 'ESC' KEY IS TREATED LIKE ANY OTHER
 80  // CHARACTER.
 90  // THE 'TRAP ESC-' STATEMENT MUST BE EXECUTED BEFORE
100  // THIS PROCEDURE IS CALLED.
110  POKE 256, 255
120  REPEAT
130   IF ESC THEN POKE 256, 27
140  UNTIL PEEK(256)<>255
150  A$:=CHR$(PEEK(256))
160 ENDPROC GET_CHR_ESC
```

Comments:

1. The value of ⟨expression⟩ must be a real or integer number greater than or equal to 0 and less than or equal to 65535. The result will be of integer type and greater than or equal to 0 and less than or equal to 255.
2. Also see 'POKE'

Type:

Machine language function

Purpose:

To set the contents of a memory position determined by an arithmetic expression.

Syntax:

POKE⟨expression1⟩,⟨expression2⟩

Execution:

The value of ⟨expression1⟩ and ⟨expression2⟩ is evaluated and rounded if necessary. The contents of the memory address corresponding to ⟨expression1⟩ is set to the value of ⟨expression2⟩.

Example:

```
10 DIM B$ OF 1
20 EXEC GET_CHARACTER(B$)
30 PRINT B$
10 PROC GET_CHARACTER(REF A$)
20  // GET KEYBOARD INPUT WITHOUT ECHO ON THE SCREEN
30  // THE 'ESC' KEY WORKS IN THE NORMAL WAY
40  POKE 256, 255
50  REPEAT
60  UNTIL PEEK(256)<>255
70  A$:=CHR$(PEEK(256))
80 ENDPROC GET_CHARACTER
```

Comments:

1. The value of ⟨expression1⟩ must be a real or integer number greater than or equal to 0 and less than or equal to 65535 and the value of ⟨expression2⟩ must be a real or integer number greater than or equal to 0 and less than or equal to 255.
2. Also see 'PEEK'.

Type:

Arithmetic function

Purpose:

To determine whether one string is contained in another and if so, where it is placed.

Syntax:

POS(⟨string expression1⟩,⟨string expression2⟩)

Execution:

It is checked, character by character, whether ⟨string expression1⟩ is contained in ⟨string expression2⟩. If it is, the result of the function is an integer, stating in which character position of ⟨string expression2⟩ that ⟨string expression1⟩ starts.

Example:

```
10 DIM A$ OF 25
20 DIM B$ OF 25
30 INPUT "FIRST STRING:  ":A$
40 INPUT "SECOND STRING: ":B$
50 C#:=POS(A$,B$)
60 PRINT C#
```

Comments:

1. If ⟨string expression1⟩ is an empty string, the function returns the result 1.
2. If ⟨string expression1⟩ is not contained in ⟨string expression2⟩, the function returns the result 0.
3. The result of the function is always of integer type.

Type:

Statement, command

Purpose:

To display data on an output device.

Syntax:

PRINT [⟨list of expressions⟩]

Execution:

The ⟨list of expressions⟩ consists of variables, constants and literals the values of which are output to the default output device.

Examples:

```
100 PRINT "THE RESULT IS: "; A
100 PRINT TAB(15); A, B
```

Comments:

1. The single elements of ⟨list of expressions⟩ are separated by commas or semicolons. If two elements are separated by a semicolon, the second element is printed immediately after the first one, while a space is inserted after an arithmetic expression. Separating two elements by a comma the second element is printed at the start of the next print-zone.
   When loading COMAL-80 the width of the print-zones is set to 0 characters.
   The width of the print-zones may be changed by 'TAB:= ⟨arithmetic expression⟩' executed as a statement or a command for which ⟨arithmetic expression⟩ is rounded to integer greater than or equal to 0.
   The rules for semicolon and comma also are valid after the last element in ⟨list of expressions⟩, as the impact is carried onto the first element of the next 'PRINT' statement.
   When ⟨list of expressions⟩ ends without a comma or semicolon, the execution of the statement ends by a change to a new line.
   This also happens if ⟨list of expressions⟩ is omitted.
2. If the remaining space on the actual line is too short to contain the next print element, it is printed from the start of the following line.

3. Switching between the output devices is done by executing a 'SELECT OUTPUT' statement.
4. (expression) being arithmetic and representing the number of character positions from the left, the function 'TAB((expression))' tabulates to the wanted character position.
   For more details also see 'TAB'.
5. During programming 'PRINT' may be substituted by ';'. In program listings 'PRINT' is used.

Type:

Statement

Purpose:

To write data in the ASCII format into a data file.

Syntax:

PRINT FILE (file No.)[,(rec. No.)]:(list of expressions)

Execution:

The values of the expressions in (list of expressions) are written in the file indicated by (file No.).

Examples:

```
100 PRINT FILE 0,RECNO: A$, B, C+D

100 DIM A$ OF 5
110 A$:="##.##"
120 PRINT FILE 3: USING "##.##": A, B, C^2
130 PRINT FILE 4: USING A$: D
```

Comments:

1. Before meeting the 'PRINT FILE (USING)' statement, a file must be opened and connection between (file name) and the (file No.) used in the 'PRINT FILE (USING)' statement must be established by the use of the 'OPEN FILE' statement or command, and type 'WRITE' or 'RANDOM'.
2. (rec. No.) is only stated for 'RANDOM' files and is an arithmetic expression which may be rounded to integer if necessary and which designates the number of the logical record of the file, which is to be utilized.
3. (file No.) is an arithmetic expression.
4. The elements in (list of expressions) are separated by commas or semicolons, similar to the syntax of 'PRINT' and 'PRINT USING'.
5. 'PRINT FILE' and 'PRINT FILE USING' perform similar to 'PRINT' and 'PRINT USING' the only difference being the destination of the output.
   The syntax for 'PRINT FILE USING' is obtained by substituting (list of expressions) in the above syntax with:
   USING (string expression):(list of expressions)
6. During programming 'FILE' and '#' are interchangeable. In program listings 'FILE' is used.
7. During programming 'PRINT' may be substituted by ';'. In program listings 'PRINT' is used.

Type:

Statement

Purpose:

To print text strings and/or numbers by use of a specified format

Syntax:

PRINT USING (string expression):(list of expressions)

Execution:

The text string specified in (string expression) is transferred character by character onto the output device, as string expressions and/or arithmetic expressions from (list of expressions) are inserted where marked '#'.

Examples:

```
100 PRINT USING "THE RESULT IS ###.##": A

 10 DIM A$ OF 6
 20 A$:="##.###"
 30 PRINT USING A$: B
```

Comments:

1. The individual characters in (string expression) have the following impact:
   '#' character position and sign.
   '.' decimal point if surrounded by '#'.
   '+' preceding plus, when '#' follows immediately after.
   '-' preceding minus, when '#' follows immediately after.
   All other characters are transferred unchanged.
2. A format starting with '+' will assign space for signs and the sign will be printed for both negative and positive values.
3. A format starting with '-' will assign space for signs but it will be printed for negative values only.
4. For text strings a preceding '+' or '-' will be equal to '#'.
5. If an arithmetic value contains too many digits to be printed in the specified format, the position is filled with '*'. If an arithmetic value contains more decimals than specified in the format, a rounding is automatically done.
6. Text strings always start at the very left within the format. If a string is too long, the necessary number of characters is deleted from the right. When a text string is too short, the rest of the format is filled with spaces.

7. When there are no more expressions in ⟨list of expressions⟩ the execution of the 'PRINT USING' statement is terminated. If ⟨list of expressions⟩ contains more expressions than stated in ⟨string expression⟩, the formats within are again used from the left.
8. Ending the 'PRINT USING' statement with a comma, the next printout will happen immediately after the output produced by the 'PRINT USING' statement. Otherwise the execution of the 'PRINT USING' statement will conclude by a change to a new line.
9. The 'PRINT USING' statement may be used for writing in a data file following exactly the same rules as described for the 'PRINT FILE' statement.
10. During programming 'PRINT' may be substituted by ';'. In program listings 'PRINT' is used.

Type:

Statement

Purpose:

To define a sub-program (a procedure)

Syntax:

```
PROC (name) [[REF] (variable) [(dim)]] [CLOSED]
.
.
.
ENDPROC (name)
```

Execution:

Meeting a 'PROC' statement the program section is skipped up to and including the corresponding 'ENDPROC' statement, and will be executed when the procedure is called by a connected 'EXEC' statement, only.

Examples:

```
10 PROC ERROR(N#) CLOSED
20  GLOBAL CC#, ERR_, ERRORS#
30  PRINT "*****";SPC$(CC#-9);"^";N#
40  ERR_:=FNINCLUDE(ERR_,N#+1); ERRORS#:+1
50 ENDPROC ERROR

PROCEDURE HEADINGS ONLY:
10 PROC XYZ(A,B,REF C$) CLOSED
10 PROC ZYX(REF A#(,,), REF C(), D$)
10 PROC YZX(REF D$(,,), REF E#, REF C) CLOSED
```

Comments:

1. The 'PROC' statement may not be used within the following statements:
   - Conditional statements
   - 'CASE' statements
   - Repeating statements
   - 'PROC' statements
   - Function declarations
2. A procedure may call other procedures, and even itself (recursion).
3. (variable) contains the names of the formal parameters which, when called by the procedure, will receive values from the actual parameters in the corresponding 'EXEC' statement.

4. The changes happening to a parameter in a procedure are local unless it is stated by 'REF' that the changes must affect the actual parameter, too.
5. 'REF' may be stated for simple arithmetic or string variables.
   'REF' must be stated for all array variables.
6. Array variables must be followed by a dimension definition consisting of commas in paranthesis, corresponding the dimension -1, i.e. for 3-dimensional arrays the paranthesis contains 2 commas whereas a vector is followed by an empty paranthesis.
7. If the procedure by instruction is declared 'CLOSED' all variable names are local and may be used for other purposes outside the procedure. This function may be declared void for one or more variables by the 'GLOBAL' statement.

Type:

Statement, command

Purpose:

To stop the COMAL-80 interpreter and return to the environment which called it.

Syntax:

QUIT

Execution:

Under CP/M, a warm boot is performed, thus transferring control to the CCP.

Examples:

```
100 QUIT
    QUIT
```

Type:

Statement, command

Purpose:

To set a random startpoint for the 'RND' functon.

Syntax:

RANDOM
RANDOMIZE

Execution:

A Z-80 CPU has a built-in counter which is read and the found value is used as the seed for the algorithm presenting a random value at the call of the 'RND' function.

Examples:

```
100 RANDOM
    RANDOM
```

Comments:

1. 'RANDOM' and 'RANDOMIZE' are interchangable. In program listings 'RANDOM' is used.
2. The counter works constantly when the the CPU is active. Its clock frequency is around 500 KHz when the CPU clock frequency is 2.5MHz.
3. If 'RANDOM' is not found in a program calling the 'RND' function, any execution of the program will give the same sequence of random numbers.

Type:

Statement

Purpose:

To assign values from the data list to variables.

Syntax:

READ (variable list)

Execution:

The single elements of (variable list) are assigned values from the data list. This is done in sequence from left to right.

Examples:

```
10 DIM FIRST_NAME$ OF 10
20 DIM FAMILY_NAME$ OF 10
30 DATA "JOHN", "DOE", 10
40 READ FIRST_NAME$, FAMILY_NAME$
50 PRINT FIRST_NAME$+" "+FAMILY_NAME$
60 READ AGE
70 PRINT AGE; "YEAR"
```

Comments:

1. If the type of value does not correspond to that of the stated variable or if the data list is empty, the program execution is stopped followed by an error message.
2. Assigning values to a string variable, follows the same rules as given for 'LET' statements.
3. Also see the 'DATA' statement.

Type:

Statement

Purpose:

To read data from a binary datafile written by the 'WRITE FILE' statement.

Syntax:

READ FILE ⟨file No.⟩ [,⟨rec No.⟩]:⟨variable list⟩

Execution:

The values of the variables in ⟨variable list⟩ are read from the file contained in ⟨file No.⟩.

Examples:

```
100 READ FILE 5,REC_NO: A
100 READ FILE 3: A, B, C
```

Comments:

1. Before meeting the 'READ FILE' statement a file must be opened and the connection established between the stated file name and the used ⟨file No.⟩ of the 'READ FILE' statement. This is done by the 'OPEN FILE' statement or command and type 'READ' or 'RANDOM'.
2. The ⟨rec No.⟩ is only used in 'RANDOM' files and is an arithmetic expression which is rounded to integer if necessary.
3. ⟨file No.⟩ is an arithmetic expression.
4. ⟨variable list⟩ may contain all variable types. Arrays are read in total if no indices are stated.
5. The elements of ⟨variable list⟩ are separated by commas.
6. During programming 'FILE' and '#' are interchangeable. In program listings 'FILE' is used.

Type:

Statement, command

Purpose:

To check that all disk files are closed.

Syntax:

RELEASE [(device)]

Execution:

It is checked whether all disk files are closed.

Examples:

```
100 RELEASE ""
100 RELEASE "DK1:"
100 RELEASE "DK"+DISK$+":"
    RELEASE
    RELEASE DK1:
```

Comments:

1. Under CP/M, the (device) indication is not used, but if it is given, it must be the name of a disk drive.
2. If a disk file is open the execution is terminated and an error message displayed.

Type:

Statement

Purpose:

To allow for insertion of explaining text in a COMAL-80 program.

Syntax:

```
//
REM
!
```

Execution:

The 'REM' statement is skipped during program execution.

Examples:

```
10 //PROGRAM TO CALCULATE
20 REM POLYNOMIAL
30 ! 30/10/1980
40 OPEN FILE 4,"TEST",READ //OPEN DATA FILE
```

Comments:

1. During programming 'REM', '//', and '!' are interchangeable. In program listings '//' is used.
2. All statements can be followed by a comment.

Type:

Statement, command

Purpose:

To change the name of a file on the background storage.

Syntax:

RENAME ⟨old file name⟩,⟨new file name⟩

Execution:

The operating system of the computer is called and parameters for 'old name' and 'new name' are used.

Examples:

```
220 RENAME "DK1:FIL.CML","DK1:FIL.BAK"
    RENAME DK1:FIL.CML,DK1:FIL.BAK
    RENAME FIL.CML,FIL.BAK
```

Comments:

1. ⟨old file name⟩ must be one existing on the stated device.
2. If no device is stated the statement/command is carried out on the current default device.
3. If the ⟨new file name⟩ is already present, this is reported and the statement/command is terminated.
4. If a device description is contained in one of the names the same device indication must be part of the other name.

Type:

Command

Purpose:

To renumber program lines and move areas of programs.

Syntax:

RENUM [[⟨start line⟩:⟨end line⟩,]⟨start⟩[,⟨step⟩]]

Execution:

If only an area of a program is to be renumbered it is checked whether there is sufficient room between the two line numbers before and after the place of the new numbers. If not, the execution is stopped followed by an error message.
If there is room enough, the new line numbers are calculated and stored. The program is checked and all references ('GOTO', 'GOSUB', etc.) are updated.
Finally, the old line numbers are deleted.

Examples:

```
RENUM
RENUM 15
RENUM 15,3
RENUM 20:90,310,1
```

Comments:

1. If ⟨step⟩ is not stated, default 10 is used.
2. If ⟨start⟩ is not stated, default 10 is used.
3. ⟨start line⟩ and ⟨end line⟩ are used when only a section of a program is renumbered and specifies the first and last line number to renumber. In this case ⟨start⟩ specifies the first new line number and ⟨step⟩ the new step between line numbers. In this way a program section optionally can be moved to any place in a program, if there are enough free line numbers, starting in ⟨start⟩ and using the indicated ⟨step⟩, before the next original line number, to contain the program section. No overwriting and no mixing can take place.
4. If ⟨start line⟩:⟨endline⟩, is not stated the total program is renumbered.

Type:

Statement

Purpose:

To repeat the execution of a program section until the condition contained in the 'UNTIL' statement is fulfilled.

Syntax:

```
REPEAT
.
.
.
UNTIL (logical expression)
```

Execution:

Meeting the 'UNTIL' statement the value of the (logical expression) is calculated. If this is true, execution resumes from the first executable statement following the 'UNTIL' statement. If the (logical expression) is false the program continues from the first executable statement following the 'REPEAT' statement.

Example:

```
 10 DIM A$ OF 1
 20 DIM B$ OF 25
 30 PRINT "THE PROGRAM IS STOPPED BY"
 40 PRINT "PRESSING THE 'ESC' KEY"
 50 TRAP ESC-
 60 REPEAT
 70  INPUT "WRITE A LETTER: ": A$,
 80  B$:=B$+A$
 90 UNTIL ESC
100 PRINT "YOU WROTE: "; B$
```

Comments:

1. A program section surrounded by 'REPEAT... UNTIL' is executed at least once.

Type:

Statement

Purpose:

To move the pointer of the data list, enabling a total or partial re-reading of the data list.

Syntax:

```
RESTORE (line number)
RESTORE (name)
RESTORE
```

Execution:

The pointer of the data list is set on the first constant in the stated line, or the first constant at all if no line is specified.

Example:

```
 10 LABEL AGAIN
 20 RESTORE DATA2
 30 READ X
 40 PRINT X
 50 DATA 47
 60 RESTORE 50
 70 READ X
 80 PRINT X
 90 GOTO AGAIN
100 LABEL DATA2
110 DATA -47
```

Comments:

1. If the 'RESTORE' statement contains a line number, the corresponding line must contain a 'DATA' statement.
2. If the 'RESTORE' statement contains a name, the statement immediately following the label statement defining that label must contain a 'DATA' statement.
3. If the 'RESTORE' statement contains neither a line number nor a name, the pointer is set on the first constant of the first 'DATA' statement.

Type:

Arithmetic function.

Purpose:

To create a pseudo-random number.

Syntax:

RND[(⟨expression1⟩,⟨expression2⟩)]

Execution:

Based on the seed (which can be changed by the 'RANDOM' statement/command) or the latest random number, a new is generated.

Example:

```
100 A:=RND
100 B:=RND(-5,17)
```

Comments:

1. Any execution of a program will give the same sequence of random figures unless a 'RANDOM' statement has been executed earlier in the program.
2. Omitting the two limits ⟨expression1⟩ and ⟨expression2⟩ a random real figure is created in the open interval of 0 to 1
3. If ⟨expression1⟩ and/or ⟨expression2⟩ is not an integer, rounding is done.
4. If limits are stated, the result will always be an integer in the closed interval from ⟨expression1⟩ to ⟨expression2⟩.

Type:

Arithmetic function

Purpose:

To convert an expression of real type to integer type.

Syntax:

ROUND((expression))

Execution:

(expression) being arithmetic is rounded and the result converted to integer type.

Example:

```
10 INPUT A
20 B#:=ROUND(A)
30 C:=ROUND(A)
40 PRINT B#, C
50 PRINT ROUND(5.72)
60 PRINT ROUND(-5.72)
```

Comments:

1. Rounding is done to the nearest integer. If the number has the same distance to two integers, the one with the highest absolute value is chosen.
2. (expression) is of real type. The result is of integer type. Note that an integer can be assigned to a real variable.
3. Also see the 'INT' and 'TRUNC' functions.

Type:

Command

Purpose:

To start the execution of a program.

Syntax:

RUN [(line number)]

Execution:

COMAL-80 is brought to a well-defined start position which among others, closes all files left open from a possible previous execution and initializes the variable area.
Thereafter, a special prepass checks whether the program contains structures (FOR...NEXT, LOOP...ENDLOOP, etc.) and references (EXEC, LABEL, etc.) and the internal representation of such statements is extended by information increasing the working speed.
Finally, the program execution is started at the stated line number.

Examples:

```
RUN
RUN 230
```

Comments:

1. Omitting (line number) the program starts at the lowest line number.

Type:

Command

Purpose:

To store programs on the background storage in the internal (binary) format as that of the program in the working storage of the computer.

Syntax:

SAVE (file name)

Execution:

The operating system of the computer is called giving information on (file name) and the area of the storage to be transferred.

Examples:

```
SAVE TEST
SAVE DK1:TEST
```

Comments:

1. Enabling a program to be called by the 'CHAIN' statement it must be stored by the 'SAVE' command.
2. Programs stored by the 'SAVE' command may be re-read by the 'LOAD' command.
3. The internal format may be different on the various versions of COMAL-80. Consequently, a program cannot always be stored by the 'SAVE' command in one version and read by the 'LOAD' command in an other version. Programs to be exchanged or stored for longer periods of time should therefore be stored by the 'LIST' command.
4. If (file name) is already on the device in question this is reported and the user receives the option to continue and have the old file deleted, or stop ('RETURN/ESC').
5. The extension '.CSB' is always supplied by the COMAL-80 system and cannot be stated by the user.

Type:

Statement, command

Purpose:

To specify a new default device/file for printout from the 'PRINT' and 'PRINT USING' statements.

Syntax:

SELECT OUTPUT ⟨string expression⟩

Execution:

Internal pointers in the COMAL-80 system switch to select the specified printout device/file.

Examples:

```
220 SELECT OUTPUT "LPO:"
220 SELECT OUTPUT "DK1:TEKST"
220 SELECT OUTPUT "TEKST"
220 SELECT OUTPUT "DS:"
    SELECT OUTPUT "LP:"
```

Comments:

1. Every time the program execution is started by the 'RUN' command the console is chosen as default output file.
   During program execution a new default file may be chosen by specifying the name of the peripheral or a file by ⟨string expression⟩.
   When program execution is terminated, either because it is stopped by pressing the 'ESC' key, or because it is finished, the terminal is again chosen as default output file.

Type:

Arithmetic function

Purpose:

Returns the sign of an arithmetic expression.

Syntax:

SGN(⟨expression⟩)

Execution:

⟨expression⟩ being arithmetic is calculated. If the result is greater than 0 the function returns the value 1. If the result equals 0, 0 is returned, and if the result is less than 0, -1 is returned.

Examples:

```
10 INPUT "WRITE A NUMBER: ": A
20 ON SGN(A)+2 GOTO 30,50,70
30 PRINT "A<0"
40 STOP
50 PRINT "A=0"
60 STOP
70 PRINT "A>0"
80 STOP
```

Type:

Trigonometric function

Purpose:

Returns the sine of an expression.

Syntax:

SIN(⟨expression⟩)

EXECUTION:

The sine of ⟨expression⟩ for which ⟨expression⟩ is in radians is calculated.

Examples:

```
10 INPUT A
20 PRINT SIN(A)
```

Comments:

1. ⟨expression⟩ is an arithmetic expression of real or integer type. The result will always be real.

Type:

Command

Purpose:

To display the size of the used area of the working storage of the computer.

Syntax:

SIZE

Execution:

The amount of working storage used is displayed on the terminal as well as how much space is left, and how much is used for variables.

Example:

```
SIZE
```

Comments:

1. The figures displayed indicate the number of bytes.
2. The space consumption for variables is not valid before program execution, and is stated only for variables dimensioned or in use during the latest execution.
3. The size of COMAL-80 is not displayed.

Type:

String function

Purpose:

To create a string consisting of spaces, the number of which is stated by an arithmetic expression.

Syntax:

SPC$(⟨expression⟩)

Execution:

⟨expression⟩ being arithmetic is calculated and rounded if necessary. Then a string containing that number of spaces is created.

Example:

```
10 INPUT A
20 PRINT SPC$(3*5),A
```

Comments:

1. ⟨expression⟩ must be greater than or equal to 0.

Type:

Arithmetic function

Purpose:

To calculate the square root of an arithmetic expression.

Syntax:

SQR(⟨expression⟩)

Execution:

The square root of ⟨expression⟩ being greater than or equal 0 is calculated.

Example:

```
10 INPUT A
20 PRINT SQR(A)
```

Comments:

1. ⟨expression⟩ being arithmetic is of real or integer type. The result will always be real.
2. If ⟨expression⟩ is less than 0 the execution is stopped followed by an error message. If these are inhibited by the 'TRAP ERR-' statement the system variable 'ERR' is set true (not equal to 0) and the square root is calculated from the expression:

   SQR(ABS(⟨expression⟩))

Type:

Statement

Purpose:

To stop the execution of a program.

Syntax:

STOP

Execution:

The program execution stops and the following is displayed on the screen:

STOP IN LINE nnnn

in which nnnn states the line number of the 'STOP' statement.

Example:

```
540 STOP
```

Comments:

1. The 'STOP' statement is normally used to stop the execution of a program in other lines than the last.
2. The program execution may be resumed by using the 'CON' command.

Type:

String function

Purpose:

To convert an arithmetic expression into a string.

Syntax:

STR$(<expression>)

Execution:

The arithmetic expression is calculated and converted to a string containing the characters which would be output if the value were printed by a 'PRINT' statement.

Example:

```
10 DIM B$ OF 7
20 INPUT "WRITE A NUMBER": A
30 B$ := STR$(A*1.5)
40 PRINT B$
```

Type:

Command, statement, (system variable)

Purpose:

To establish a new print-zone width by assigning this value to the system variable 'TAB'.

Syntax:

TAB:=(arithmetic expression)

Execution:

The system variable 'TAB' is assigned the value of (arithmetic expression) which is rounded if necessary.

Examples:

```
100 TAB:=8
100 TAB=X*Y+3
    TAB=12
```

Comments:

1. Loading COMAL-80, 'TAB' is assigned the value of 0. This value can be changed only by the use of a 'TAB' statement or command.
2. It is not possible to read the value of 'TAB'.
3. The 'NEW' command does not change the value of the system variable 'TAB'.
4. See 'PRINT'
5. During programming ':=' and '=' are interchangeable. In program listings ':=' is used.

Type:

Print function

Purpose:

In connection with a 'PRINT' statement to tabulate to the character position before the next printout.

Syntax:

TAB(⟨expression⟩)

Execution:

The arithmetic expression is calculated and if necessary rounded. The result defines the start position of the next printout.

Example:

```
100 PRINT TAB(10),"THE RESULT IS: ",RESULT
```

Comments:

1. TAB(⟨expression⟩) can be used in connection with 'PRINT' statements only.
2. ⟨expression⟩ is an absolute value counted from the left side margin of the output unit.
3. If the last printout before the 'TAB(⟨expression⟩)' has passed the specified position, the program execution is stopped by an error message.
4. ⟨expression⟩ being arithmetic must evaluate to a value greater than or equal to 1 and less than or equal to the maximum number of characters allowed in the width of the output device.

Type:

Trigonometric function

Purpose:

To calculate the tangent of an arithmetic expression.

Syntax:

TAN(⟨expression⟩)

Execution:

The tangent of ⟨expression⟩ which is in radians is calculated.

Example:

```
10 INPUT A
20 PRINT TAN(A)
```

Comments:

1. ⟨expression⟩ being arithmetic is of real or integer type. The result will always be real.

TYPE:

Statement, command

Purpose:

To change the normal system action on a non-fatal error.

Syntax:

```
TRAP ERR-
TRAP ERR+
```

Execution:

During a normal program execution, any error will stop the program and create an error message. However, a number of errors can be bypassed in a well-defined manner.
In such cases a program interruption may be avoided by the use of a 'TRAP ERR-' statement, before the error arises. In this case, the system variable 'ERR' will be assigned a value equal to the error number, which in all tests will be considered true because it is different from 0. The program execution will then continue.

Example:

```
 10 INIT "", FILENAME$
 20 TRAP ERR-
 30 OPEN FILE 0, "XPLOCOMM", READ
 40 TRAP ERR+
 50 IF NOT ERR THEN
 60  INPUT FILE 0: DEFAULT_FILENAME$
 70 ELSE
 80  DEFAULT_FILENAME$:="XPLOPROG"
 90 ENDIF
100 CLOSE
```

Comments:

1. The execution of a program starts by assigning the value of false (= 0) to the system variable 'ERR'. When a 'TRAP ERR-' statement has been executed, a non-fatal error assigns its error number to 'ERR' and it retains this value until its status is checked. Immediately after a such check, 'ERR' is assigned the value of false.
   Normally COMAL-80 sets a variable true by assigning it the value of 1, but in this case the error number is used.
   The error numbers are further described in appendix C.
2. By executing a 'TRAP ERR+' statement, the system returns to normal error handling.

TYPE:

Statement, command

Purpose:

To change the system action to a press on the 'ESC' key.

Syntax:

```
TRAP ESC-
TRAP ESC+
```

Execution:

During normal program execution it is checked, before each statement, whether the 'ESC' key has been pressed. In the affirmative the program execution is stopped.
If a 'TRAP ESC-' statement has been executed, this function is blocked and the system variable 'ESC' is instead assigned the value of true (= 1) when 'ESC' is pressed.

Example:

```
10 TRAP ESC-
20 REPEAT
30  PRINT "THE 'ESC' KEY IS NOT PRESSED"
40 UNTIL ESC
50 TRAP ESC+
60 PRINT "THE 'ESC' KEY WAS PRESSED"
```

Comments:

1. Starting program execution the system variable 'ESC' is assigned the value of false (= 0). If a 'TRAP ESC-' statement is executed and the 'ESC' key pressed after that, the program execution continues but the system variable 'ESC' is assigned the value of true (= 1) and keeps this value until its status is checked.
   Immediately after the value is used, 'ESC' is again assigned the value of false (= 0).
2. The system returns to normal handling of the 'ESC' key when a 'TRAP ESC+' statement is executed.

Type:

System constant

Purpose:

Mainly to assign a boolean variable the value of true.

Syntax:

TRUE

Execution:

Returns the value 1.

Example:

```
 10 // PRIME
 20 //
 30 DIM FLAGS#(0:8190)
 40 SIZE1:=8190
 50 //
 60 COUNT:=0
 70 MAT FLAGS#:=TRUE
 80 //
 90 FOR I:=0 TO SIZE1 DO
100  IF FLAGS#(I) THEN
110   PRIME:=I+I+3
120   K:=I+PRIME
130   WHILE K(=SIZE1 DO
140    FLAGS#(K):=FALSE
150    K:+PRIME
160   ENDWHILE
170   COUNT:+1
180  ENDIF
190 NEXT I
200 PRINT "TOTAL NUMBER OF PRIMES: ",COUNT
```

Type:

Arithmetic function

Purpose:

To convert an expression of real type to an integer.

Syntax:

TRUNC((expression))

Execution:

(expression) being arithmetic is evaluated and the result converted to integer type while disregarding any decimals.

Examples:

```
100 A=TRUNC(5.72)
100 A:=TRUNC(A/B)
```

Comments:

1. (expression) is of real type.
   The result is of integer type.
2. Also see the 'ROUND' and 'INT' functions.

Type:

Command

Purpose:

To assign the background storage device which will be considered the default device.

Syntax:

UNIT (device)

Execution:

The internal pointers are updated to point at the stated device.

Examples:

```
100 UNIT "DK1:"
    UNIT DK1:
```

Comments:

1. (device) is stated as 2 letters, describing the type of background storage device, followed by the unit number and a colon.

Type:

String function.

Purpose:

To convert a real number of string type to a number of real type.

Syntax:

VAL((string expression))

Execution:

The real number in (string expression) is converted to a number of real type.

Example:

```
10 DIM A$ OF 5
20 A$:="32.34"
30 PRINT VAL(A$)
```

Comments:

1. If (string expression) does not contain a well-formed real or integer number, the program execution is stopped with an error message.
2. Also see the 'IVAL' function.

Type:

Machine code function.

Purpose:

To find the absolute address in the memory at which a variable is stored.

Syntax:

VARPTR ((variable))

Execution:

The decimal, absolute address in the memory, in which the first byte af the variable (variable) is stored, is found.

Example:

```
10 INPUT A
20 PRINT VARPTR(A)
```

Comments:

1. The result states where the first byte of the variable is stored. The remainder of the bytes are on the locations following.
   Integers take 2 bytes of which the lower part of the number is first.
   Real numbers take 4 bytes in the 7-digits version.
   Real numbers take 8 bytes in the 13-digits version.
   For string variables the first 2 bytes state the length and the string is then stored consecutively.
2. The result is of real type.
3. The variable may be an array with or without indices. If no indices are stated, the address of the first element of the array is delivered.
4. WARNING: In one situation a variable is moved after it has been allocated storage, thus changing its address. This occurs upon exit from a non-closed procedure to all variables that have been encountered and allocated storage for the first time during the current call of the procedure.

Type:

Statement

Purpose:

To repeat the execution of a program section until the condition contained in the 'WHILE' statement is fulfilled.

Syntax:

```
WHILE <logical expression>
.
.
.
ENDWHILE
```

Execution:

Meeting the 'WHILE' statement the value of the <logical expression> is calculated. If this is true, execution resumes from the first executable statement following the 'WHILE' statement. If the <logical expression> is false the program continues from the first executable statement following the 'ENDWHILE' statement.

Example:

```
10 OPEN FILE 0,"DATA",READ
20 WHILE NOT EOF(0) DO
30  READ FILE 0: INDEX, NUMBER#, TEXT$
40 ENDWHILE
```

Type:

Statement

Purpose:

To write data in the binary format into a data file.

Syntax:

WRITE FILE ⟨file No.⟩ [,⟨rec. No.⟩]: ⟨variable list⟩

Execution:

The values of the variables in ⟨variable list⟩ are written in the file contained in ⟨file No.⟩.

Examples:

```
100 WRITE FILE 7,REC_NO: A, B, C
100 WRITE FILE 3: A$, B#, C
```

Comments:

1. Before meeting the 'WRITE FILE' statement, a file must be opened and connection between ⟨file name⟩ and the ⟨file No.⟩ used in the 'WRITE FILE' statement must be established by the use of the 'OPEN FILE' statement or command, and type 'WRITE' or 'RANDOM'.
2. ⟨rec. No.⟩ is only stated at 'RANDOM' files and is an arithmetic expression which may be rounded to integer if necessary.
3. ⟨file No.⟩ is an arithmetic expression.
4. ⟨variable list⟩ may contain all variable types. If an array variable is stated without indices, the whole array is written.
5. The elements in ⟨variable list⟩ are separated by commas.
6. During programming 'FILE' and '#' are interchangeable. In program listings 'FILE' is used.

MODIFYING COMAL-80

COMAL-80 is a very interactive program in the way that it tries to help the user to a correct program by displaying error messages and moving the cursor to points, where there are problems. It is therefore necessary that the connected terminal supports functions like 'erase to end of line', 'erase to end of screen', cursor addressing and a few more.

Unfortunately, the specifications for CP/M do not include a description of how these functions should be implemented and many different methods are used.

To overcome this problem, the source code for the screen driver is shown in appendix B, and it will normally be possible to change this driver, so that most CRT-terminals can be used.

It is not recommended to use printing terminals like teletypes.

The necessary changes normally are very easy to do in a few minutes by replacing control characters in a table with the actual ones.

STEP BY STEP GUIDE.

1. Make a copy of the received disk, remove this disk from the computer and store it in a safe place. Remember, that your warranty is carried by this disk only.

2. Read the source code for the screen driver and this guide carefully.

3. Read the manual for the actual terminal and check whether it supports the functions mentioned in the table defining the control characters.

   If it does, you are in for an easy job. Carry on.

   If it does not, go to step 13.

4. Go to your computer and use DDT to make the necessary changes. Depending on which version you want to change, enter

```
DDT COMAL-80.COM        or
DDT COMAL80S.COM        or
DDT COMAL80D.COM        or
DDT CMAL80DS.COM
```

   and remember which version you are working on.

5. Check whether the actual control characters the terminal wants, are the same as those shown in the control-character table placed in the hexadecimal addresses 15C7H to 15D2H.

   If they are, go to step 6.

   If not, replace the old ones by the new ones.

6. Place in address 15D3H the hexadecimal number of characters per line and in address 15D4H the hexadecimal number of lines on the screen. The original values in those two places are 28H and 18H.

7. Check, that the cursor address routine called 'GOTOXY' and placed in adresses 174FH to 1768H works in a way, that the actual terminal wants.

   'GOTOXY' firstly sends an 'ESC' character, then a '=', then the line number and last the character number adding hexadecimal 20H to the latter two.

   If the terminal needs something else, change 'GOTOXY' as necessary. If the new routine is larger than the old one, place the rest (or the whole routine) in the free space starting in address 17E2H.

8. COMAL-80 expects that the terminal is equipped with an 'ESC' key sending the hexadecimal code '1BH'. If this is not the case with the actual terminal, change the following two places:

   1894H and 1AC3H

   to the new code or the code for a suitable key. This key is very important as it stops everything and it is best to use a key, which is easy to find without looking at the keyboard.

9. Ten other keys can be redefined. These are:

FUNCTION	ORIGINAL VALUE	ORIGINAL CHARACTER
CURSOR RIGHT	1DH	control]
CURSOR LEFT	1CH	control \
INSERT	01H	control A
DELETE	13H	control S
BACKSPACE	08H	control H
CURSOR TO START OF LINE	15H	control U
CURSOR TO END OF LINE	05H	control E
CURSOR 8 STEP FORWARD	09H	control I
CURSOR 8 STEP BACKWARD	02H	control B
DELETE TO END OF LINE	0BH	control K

These functions can be related to new keys simply by inserting the new code in the following addresses:

CURSOR RIGHT	1897H
CURSOR LEFT	1881H
INSERT	18ECH
DELETE	18B1H
BACKSPACE	192DH
CURSOR TO START OF LINE	195CH
CURSOR TO END OF LINE	1976H
CURSOR 8 STEP FORWARD	198EH
CURSOR 8 STEP BACKWARD	19BAH
DELETE TO END OF LINE	19E7H

These changes affect only the transmission from the keyboard to the computer and have no influence on the transmission from the computer to the screen.

10. If the terminal has more than 64 characters per line, the 'CAT' command should be changed to list four files per line by changing addresses 142FH and 1464H to 04 instead of 02.

11. The last thing to do is to tell COMAL-80 how many disk drives are connected to the computer. Do this by inserting the number of disks minus one in address 145H. The original value in this address is 01H which means that COMAL-80 is prepared for 2 disks.

12. Press control-C and when CP/M has re-initialized enter:

```
SAVE 155 COMAL-80.COM          or
SAVE 110 COMAL80S.COM          or
SAVE 156 COMAL80D.COM          or
SAVE 111 CMAL80DS.COM
```

depending on which version you worked on.

13. Terminals, which do not support cursor addressing or other functions which COMAL-80 needs are a bit more complicated, as some assembler programming will be necessary.

Do not try to do these changes unless you have a relatively good knowledge of this special art.

Unfortunately, due to big differences in the way the various terminals work, it is not possible to tell exactly how the screen driver should be changed but it is possible to give some guidelines.

```
     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
     ;
     ;        SCREEN DRIVER FOR COMAL-80 V 1.8
     ;      COPYRIGHT (C) 1981 METANIC ApS DENMARK
     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
     ;
     ; ASCII NUMBERS OF SOME CONTROL CHARACTERS
     ; THESE CHARACTERS ARE USED INSIDE COMAL-80 AND MUST NOT
     ; BE CHANGED. THE ACTUAL KEYBOARD CHARACTERS DO NOT
     ; AFFECT THIS TABLE.
     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
             PSECT   ABS
15B2         ORG     15B2H            ; VERSION 1.8 ONLY
     ;
001B ESC     EQU     1BH              ; ESCAPE CHARACTER
000D CR      EQU     0DH              ; CARRIAGE RETURN
0008 CLEFT   EQU     08H              ; CURSOR LEFT
000C CRIGHT  EQU     0CH              ; CURSOR RIGHT
000B CUP     EQU     0BH              ; CURSOR UP
000A CDOWN   EQU     0AH              ; CURSOR DOWN
001E CHOME   EQU     1EH              ; CURSOR HOME
001F CLRLINE EQU     1FH              ; CLEAR REST OF LINE
001D CLRDISP EQU     1DH              ; CLEAR REST OF DISPLAY
001B LEADIN  EQU     1BH              ; LEAD IN CHARACTER

     ; VARIABLE ADDRESSES - THESE VARIABLES ARE PLACES IN THE
     ;   SAME ADDRESSES AS THE INITIALISATION CODE.

0108 CURSOR  EQU     108H             ; LOGICAL CURSOR ADDRESS
                                      ;   RELATIVE TO HOME POS.
                                      ;
                                      ;   ALWAYS  = CHARNO +
                                      ;   #CHRLIN*LINENO
010A CHARNO  EQU     10AH             ; X ADDRESS OF CURSOR POS.
                                      ;   IN RANGE 0..#CHRLIN-1
010B LINENO  EQU     10BH             ; Y ADDRESS OF CURSOR POS.
                                      ;   IN RANGE 0..#LINES-1.
                                      ;   HOME POS. HAS LINENO=0

010C LASTWASPRINTABLE EQU    10CH     ; FLAG THAT TELLS IF THE
                                      ;   LAST OPERATION ON THE
                                      ;   DISPLAY WAS OUTPUTTING
                                      ;   A PRINTABLE CHARACTER.
                                      ;   CALLS OF 'MOVECURSOR'
                                      ;   ARE BLIND IN THIS
                                      ;   RESPECT.
010D LASTW1  EQU     10DH             ; TEMPORARY FOR
                                      ;   'LASTWASPRINTABLE'

1C55 OPENMO  EQU     1C55H            ; VERSION 1.8 ONLY
184E CRTIN   EQU     184EH            ; VERSION 1.8 ONLY
0005 XBDOS   EQU     05H
```

```
                 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                 ;
                 ;
                 ; THIS TABLE ESTABLISHES THE CONNECTION BETWEEN COMAL-80
                 ; AND THE SCREEN_DRIVER.
                 ; IF THE SCREEN_DRIVER IS CHANGED, THIS TABLE MUST BE
                 ; CHANGED TOO, BUT THE REST OF COMAL-80 IS UNAFFECTED.
                 ;
                 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
15B2  C3D515     DSSTART         JP      XDSSTART
15B5  C3D615     DSEND           JP      XDSEND
15B8  C3D715     CLRSCREEN       JP      XCLRSCRE
15BB  C3E215     CRTOUT          JP      XCRTOUT
15BE  C36917     CHARIN          JP      XCHARIN
15C1  C37A17     MOVECURSOR      JP      XMOVECURSOR
15C4  C3AB17     PLACECURSOR     JP      XPLACECURSOR




                 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                 ;
                 ; THIS TABLE DEFINES THE CONTROLCHARACTERS FOR THE SCREEN
                 ; AS WELL AS THE SCREEN FORMAT.
                 ;
                 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
15C7  000C       CURIGHT DEFB    00,CRIGHT           ; CURSOR RIGHT
15C9  000B       CUUP    DEFB    00,CUP              ; CURSOR UP
15CB  000A       CUDOWN  DEFB    00,CDOWN            ; CURSOR DOWN
15CD  001E       CUHOME  DEFB    00,CHOME            ; CURSOR HOME
15CF  1B54       CLEAR   DEFB    LEADIN,'T'          ; CLEAR REST OF LINE
15D1  1B59       CLEARD  DEFB    LEADIN,'Y'          ; CLEAR REST OF DISPLAY
15D3  28         #CHRLIN DEFB    40                  ; CHARACTERS PR LINE
15D4  18         #LINES  DEFB    24                  ; LINES PR PAGE



                 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                 ;
                 ; PROCEDURE DSSTART     INITIALISATION PROCEDURE
                 ;
                 ; NO INPUT, NO OUTPUT
                 ;
                 ; FUNCTION:
                 ;      INITIALISATION FOR THE CRT DRIVER.
                 ;
                 ;      USED AT START-UP TIME ONLY
                 ;
                 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

15D5  C9         XDSSTART:        RET

```

```
                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                ;
                ; PROCEDURE DSEND          FINALISATION PROCEDURE
                ;
                ; NO INPUT, NO OUTPUT
                ;
                ; FUNCTION:
                ;      FINALIZATION FOR THE CRT DRIVER
                ;
                ;    USED IN CLOSING DOWN THE COMAL SYSTEM.
                ;
                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
15D6  C9        XDSEND:  RET
                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                ;
                ; PROCEDURE CLRSCREEN      CLEAR SCREEN
                ;
                ; NO INPUT, NO OUTPUT
                ;
                ; FUNCTION:
                ;      CLEARS THE DATA SCREEN AND SETS THE CURSOR IN THE
                ;    UPPER LEFT HAND CORNER.
                ;
                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
15D7            XCLRSCREEN:
15D7  21E015            LD      HL,CLRS90       ; WRITE CHOME, CLRDISP
15DA  110200            LD      DE,2
15DD  C3E215            JP      XCRTOUT
15E0  1E1D      CLRS90: DEFB    CHOME,CLRDISP
```

```
                 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                 ;
                 ; PROCEDURE CRTOUT        OUTPUT TO CRT
                 ;
                 ; INPUT:   HL : PTR TO A TEXT
                 ;          DE : THE NUMBER OF CHARACTERS IN THE TEXT
                 ;
                 ; NO OUTPUT
                 ;
                 ; FUNCTION:
                 ;       THE TEXT IS OUTPUT AT THE CURRENT CURSOR POSITION
                 ;    ON THE CRT. THE CURSOR POSITION IS UPDATED. SCROLL
                 ;    IS IMPLEMENTED. THE CONTROL CHARACTERS THAT ARE
                 ;    RECOGNISED ARE MENTIONED IN THE CONSTANTS SECTION
                 ;    IN THE BEGINNING OF THIS FILE.
                 ;
                 ; MODIFIES AF, DE, HL, BC', DE', HL'
                 ;
                 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

15E2             XCRTOUT:
15E2  7A         CRT005: LD      A,D                 ; WHILE DE <> 0 DO
15E3  B3                 OR      E
15E4  C8                 RET     Z
15E5  AF                 XOR     A
15E6  320D01             LD      (LASTW1),A          ;   LASTW1 := FALSE
15E9  7E                 LD      A,(HL)              ;   A := (HL) BITS 0-6
15EA  CBBF               RES     7,A                 ;
15EC  23                 INC     HL                  ;   HL :+ 1
15ED  1B                 DEC     DE                  ;   DE :- 1
15EE  D9                 EXX                         ;   (ALTERNATE BANK)
15EF  FE20               CP      ' '                 ;   IF A < ' ' THEN
15F1  D20B17             JP      NC,CRT075
15F4  FE0D               CP      CR                  ;     IF A = CR THEN
15F6  2023               JR      NZ,CRT020
15F8  47                 LD      B,A
15F9  3A0A01             LD      A,(CHARNO)          ;       IF CHARNO <> 0
15FC  5F                 LD      E,A
15FD  B7                 OR      A
15FE  2007               JR      NZ,CRT010
1600  3A0C01             LD      A,(LASTWASPRINTABLE);         OR NOT
1603  B7                 OR      A                   ;         LASTWASPRINTABLE
1604  C22817             JP      NZ,CRT085           ;       THEN
1607  2A0801     CRT010: LD      HL,(CURSOR)         ;         CURSOR :- CHARNO
160A  AF                 XOR     A
160B  57                 LD      D,A
160C  ED52               SBC     HL,DE
160E  220801             LD      (CURSOR),HL
1611  320A01             LD      (CHARNO),A          ;         CHARNO := 0
1614  78                 LD      A,B
1615  CD3217             CALL    CRT072              ;         NORMALWRITE(A)
1618  C3B816             JP      CRT051              ;         GOTO CURSOR_DOWN
```

```
161B  FE08      0200 CRT020: CP      CLEFT           ;     ELIF A = CLEFT THEN
161D  2033      0201         JR      NZ,CRT030
161F  CD3217    0202         CALL    CRT072          ;       NORMALWRITE(A)
1622  2A0801    0203         LD      HL,(CURSOR)     ;       CURSOR :- 1
1625  2B        0204         DEC     HL
1626  220801    0205         LD      (CURSOR),HL
1629  CB7C      0206         BIT     7,H             ;       IF CURSOR < 0
162B  2810      0207         JR      Z,CRT025        ;       THEN
162D  3AD315    0208         LD      A,(#CHRLIN)     ;         CURSOR :=
1630  3D        0209         DEC     A               ;           #CHRLIN-1
1631  6F        0210         LD      L,A             ;         CHARNO :=
1632  2600      0211         LD      H,0             ;           #CHRLIN-1
1634  220801    0212         LD      (CURSOR),HL
1637  320A01    0213         LD      (CHARNO),A
163A  C32817    0214         JP      CRT085          ;
163D  3A0A01    0215 CRT025: LD      A,(CHARNO)      ;       ELSE
1640  C6FF      0216         ADD     A,-1            ;         CHARNO :- 1
1642  3808      0217         JR      C,CRT028        ;         IF CHARNO < 0
1644  210B01    0218         LD      HL,LINENO       ;         THEN
1647  35        0219         DEC     (HL)            ;           LINENO :- 1
1648  3AD315    0220         LD      A,(#CHRLIN)     ;           CHARNO :=
164B  3D        0221         DEC     A               ;             #CHRLIN-1
164C  320A01    0222 CRT028: LD      (CHARNO),A      ;         ENDIF
164F  C32817    0223         JP      CRT085          ;       ENDIF
                0224
1652  FE0C      0225 CRT030: CP      CRIGHT          ;     ELIF A = CRIGHT THEN
1654  2038      0226         JR      NZ,CRT040
1656  21C715    0227         LD      HL,CURIGHT      ;       CONTROLWRITE(
1659  CD3D17    0228         CALL    CONWRI          ;         CURIGHT)
165C            0229 CRT032:                         ; CURSOR_RIGHT:
165C  2A0801    0230         LD      HL,(CURSOR)     ;       CURSOR :+ 1
165F  23        0231         INC     HL
1660  220801    0232         LD      (CURSOR),HL
1663  210A01    0233         LD      HL,CHARNO       ;       CHARNO :+ 1
1666  34        0234         INC     (HL)
1667  3AD315    0235         LD      A,(#CHRLIN)     ;       IF CHARNO=#CHRLIN
166A  BE        0236         CP      (HL)
166B  C22817    0237         JP      NZ,CRT085       ;       THEN
166E  3600      0238         LD      (HL),0          ;         CHARNO := 0
1670  210B01    0239         LD      HL,LINENO       ;         LINENO :+ 1
1673  34        0240         INC     (HL)
1674  3AD415    0241         LD      A,(#LINES)      ;         IF LINENO =
1677  BE        0242         CP      (HL)            ;           #LINES
1678  C22817    0243         JP      NZ,CRT085       ;         THEN
167B  35        0244         DEC     (HL)            ;           LINENO :- 1
167C  2A0801    0245         LD      HL,(CURSOR)     ;           CURSOR :-
167F  3AD315    0246         LD      A,(#CHRLIN)     ;             #CHRLIN
1682  5F        0247         LD      E,A
1683  1600      0248         LD      D,0
1685  A7        0249         AND     A
1686  ED52      0250         SBC     HL,DE
1688  220801    0251         LD      (CURSOR),HL
168B  C32817    0252         JP      CRT085          ;         ENDIF
                0253                                 ;       ENDIF
                0254
168E  FE0B      0255 CRT040: CP      CUP             ;     ELIF A = CUP THEN
1690  2022      0256         JR      NZ,CRT050
1692  21C915    0257         LD      HL,CUUP         ;       CONTROLWRITE(
1695  CD3D17    0258         CALL    CONWRI          ;         CUUP)
```

```
1698             0259 CRT042:                        ; CURSOR_UP:
1698  3A0B01     0260         LD      A,(LINENO)
169B  B7         0261         OR      A              ;       IF LINENO ) 0
169C  2813       0262         JR      Z,CRT045       ;       THEN
169E  3D         0263         DEC     A
169F  320B01     0264         LD      (LINENO),A     ;         LINENO :- 1
16A2  3AD315     0265         LD      A,(#CHRLIN)
16A5  5F         0266         LD      E,A
16A6  1600       0267         LD      D,0
16A8  2A0801     0268         LD      HL,(CURSOR)    ;         CURSOR :-
16AB  A7         0269         AND     A              ;           #CHRLIN
16AC  ED52       0270         SBC     HL,DE
16AE  220801     0271         LD      (CURSOR),HL
16B1  C32817     0272 CRT045  JP      CRT085         ;       ENDIF
                 0273
16B4  FE0A       0274 CRT050: CP      CDOWN          ;     ELIF A = CDOWN THEN
16B6  2021       0275         JR      NZ,CRT060
16B8  3E0A       0276 CRT051  LD      A,CDOWN        ; CURSOR_DOWN:
16BA  CD3217     0277         CALL    CRT072         ;       NORMALWRITE(CDOWN)
16BD  3A0B01     0278         LD      A,(LINENO)
16C0  3C         0279         INC     A
16C1  21D415     0280         LD      HL,#LINES      ;       IF LINENO (
16C4  BE         0281         CP      (HL)           ;         #LINES-1
16C5  2810       0282         JR      Z,CRT055       ;       THEN
16C7  320B01     0283         LD      (LINENO),A     ;         LINENO :+ 1
16CA  2A0801     0284         LD      HL,(CURSOR)    ;         CURSOR :+
16CD  3AD315     0285         LD      A,(#CHRLIN)    ;           #CHRLIN
16D0  5F         0286         LD      E,A
16D1  1600       0287         LD      D,0
16D3  19         0288         ADD     HL,DE
16D4  220801     0289         LD      (CURSOR),HL
16D7  184F       0290 CRT055: JR      CRT085         ;       ENDIF
                 0291
16D9  FE1E       0292 CRT060: CP      CHOME          ;     ELIF A = CHOME THEN
16DB  2015       0293         JR      NZ,CRT065
16DD  21CD15     0294         LD      HL,CUHOME      ;       CONTROLWRITE(
16E0  CD3D17     0295         CALL    CONWRI         ;         CUHOME)
16E3  210000     0296         LD      HL,0
16E6  220801     0297         LD      (CURSOR),HL    ;       CURSOR := 0
16E9  AF         0298         XOR     A
16EA  320A01     0299         LD      (CHARNO),A     ;       CHARNO := 0
16ED  320B01     0300         LD      (LINENO),A     ;       LINENO := 0
16F0  1836       0301         JR      CRT085
                 0302
16F2  FE1F       0303 CRT065: CP      CLRLINE        ;     ELIF A = CLRLINE
16F4  2008       0304         JR      NZ,CRT070      ;     THEN
16F6  21CF15     0305         LD      HL,CLEAR
16F9  CD3D17     0306         CALL    CONWRI         ;     CONTROLWRITE(
16FC  182A       0307         JR      CRT085         ;       CLEAR)
                 0308
16FE  FE1D       0309 CRT070: CP      CLRDISP        ;     ELIF A = CLRDISP
1700  C22817     0310         JP      NZ,CRT085      ;     THEN
1703  21D115     0311         LD      HL,CLEARD
1706  CD3D17     0312         CALL    CONWRI         ;       CONTROLWRITE(
1709  181D       0313         JR      CRT085         ;         CLEARD)
                 0314                                ;     ELSE
                 0315                                ;       NOTHING
                 0316                                ;     ENDIF
```

```
170B             0317 CRT075:                          ;   ELSE
                 0318                                  ;     IF A()0FFH THEN
170B  FEFF       0319          CP      0FFH
170D  280B       0320          JR      Z,CRT080
170F  CD3217     0321          CALL    CRT072          ;       NORMALWRITE(A)
1712  3E01       0322          LD      A,1
1714  320D01     0323          LD      (LASTW1),A      ;       LASTW1:=TRUE
1717  C35C16     0324          JP      CRT032          ;       GOTO CURSOR_RIGHT
171A  5F         0325 CRT080   LD      E,A                   ELSE
171B  0E02       0326          LD      C,02
171D  CDCC17     0327          CALL    BDOS            ;       BDOS.WRITE(A)
1720  3E01       0328          LD      A,1             ;       LASTW1 := TRUE
1722  320D01     0329          LD      (LASTW1),A
1725  C35C16     0330          JP      CRT032          ;       GOTO CURSOR_RIGHT
                 0331                                  ;     ENDIF
1728             0332 CRT085:                          ;   ENDIF
                 0333
1728  3A0D01     0334          LD      A,(LASTW1)      ;   LASTWASPRINTABLE :=
172B  320C01     0335          LD      (LASTWASPRINTABLE),A; LASTW1
172E  D9         0336          EXX                     ;     (MAIN BANK)
172F  C3E215     0337          JP      CRT005          ; ENDWHILE
                 0338
                 0339
                 0340
                 0341 ;
                 0342 ; PROCEDURE NORMALWRITE
                 0343 ;
                 0344 ; INPUT:         A CHARACTER
                 0345 ;
                 0346 ; NO OUTPUT
                 0347 ;
                 0348 ; FUNCTION:      OUTPUTS A ON THE CRT. ASSUMES THAT A IS A
                 0349 ;                PRINTABLE CHARACTER, CR, CURSOR_LEFT OR
                 0350 ;                CURSOR_DOWN (LINEFEED)
                 0351 ;
                 0352 ; MODIFIES AF,BC,DE,HL
                 0353 ;
1732  E5         0354 CRT072   PUSH    HL
1733  D5         0355          PUSH    DE
1734  5F         0356          LD      E,A
1735  0E06       0357          LD      C,6
1737  CDCC17     0358          CALL    BDOS
173A  D1         0359          POP     DE
173B  E1         0360          POP     HL
173C  C9         0361          RET
                 0362
```

```
                ;
                ; PROCEDURE CONTROLWRITE
                ;
                ; INPUT:  HL POINTS OUT EN ENTRY IN THE TRANSLATION TABLE
                ;         THAT STARTS A LABEL CURIGHT. THIS ENTRY CONSISTS
                ;         OF TWO BYTES. IF THE FIRST BYTE IS > 0, IT IS
                ;         WRITTEN OUT. THE SECOND BYTE IS ALWAYS WRITTEN
                ;         OUT.
                ;
                ; NO OUTPUT
                ;

173D            CONWRI:
173D  7E                LD      A,(HL)          ; GET FIRST
173E  B7                OR      A               ; SET FLAGS
173F  C44417            CALL    NZ,CONW10       ; IF NOT ZERO
1742  23                INC     HL              ; INC POINTER
1743  7E                LD      A,(HL)          ; GET SECOND
1744  E5        CONW10: PUSH    HL              ; SAVE HL
1745  D5                PUSH    DE              ; SAVE DE
1746  5F                LD      E,A             ; MAKE READY FOR CP/M
1747  0E06              LD      C,6
1749  CDCC17            CALL    BDOS            ; CALL CP/M
174C  D1                POP     DE              ; RESTORE DE
174D  E1                POP     HL              ; RESTORE HL
174E  C9                RET                     ; RETURN



                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                ;
                ; PROCEDURE GOTOXY                        POSITION CURSOR
                ;
                ; NO REGISTER INPUT OR OUTPUT
                ;
                ; FUNCTION:
                ;       THE CURSOR IS POSITIONED AT THE X, Y COORDINATES
                ; FOUND IN THE VARIABLES CHARNO AND LINENO.
                ;
                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

174F            GOTOXY:
174F  3E1B              LD      A,ESC
1751  CD3217            CALL    CRT072          ; NORMALWRITE(ESC)
1754  3E3D              LD      A,'='
1756  CD3217            CALL    CRT072          ; NORMALWRITE('=')
1759  3A0B01            LD      A,(LINENO)
175C  C620              ADD     A,32            ; OFFSET USED BY MANY TER-
                                                ; NALS
175E  CD3217            CALL    CRT072          ; NORMALWRITE(LINENO)
1761  3A0A01            LD      A,(CHARNO)
1764  C620              ADD     A,32            ; OFFSET USED BY MANY TER-
                                                ; MINALS
1766  C33217            JP      CRT072          ; NORMALWRITE(CHARNO)
```

```
                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                ;
                ; PROCEDURE CHARIN          INPUT CHARACTER
                ;
                ; NO INPUT
                ;
                ; OUTPUT: A : CHARACTER
                ;
                ; FUNCTION:
                ;       READS A CHARACTER FROM THE KEYBOARD.
                ;
                ; MODIFIES AF
                ;
                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

1769            XCHARIN:
1769  E5                PUSH    HL
176A  D5                PUSH    DE
176B  C5                PUSH    BC
176C  0E06      XCHA10: LD      C,06
176E  1EFF              LD      E,0FFH
1770  CDCC17            CALL    BDOS
1773  B7                OR      A
1774  CBBF              RES     7,A
1776  C1                POP     BC
1777  D1                POP     DE
1778  E1                POP     HL
1779  C9                RET



                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                ;
                ; PROCEDURE MOVECURSOR
                ;
                ; INPUT:  HL : NUMBER OF CHARACTERS TO MOVE THE CURSOR
                ;              (SIGNED: + FORWARDS, - BACKWARDS)
                ;
                ; NO OUTPUT
                ;
                ; FUNCTION:
                ;         MOVES THE CURSOR UNDER THE ASSUMPTION THAT NO
                ;     SCROLLING IS NECESSARY.
                ;
                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

177A            XMOVECURSOR:
177A  E5                PUSH    HL
177B  3A0A01            LD      A,(CHARNO)      ; CHARNO :+ HL
177E  5F                LD      E,A
177F  1600              LD      D,0
1781  19                ADD     HL,DE
1782  3AD315            LD      A,(#CHRLIN)
1785  5F                LD      E,A
1786  1600              LD      D,0
1788  3A0B01            LD      A,(LINENO)
```

```
178B  A7        MOVE10: AND    A                   ; REPEAT
178C  3C                INC    A                   ;   LINENO :+ 1
178D  ED52              SBC    HL,DE               ;   CHARNO :- 80
178F  F28B17            JP     P,MOVE10            ; UNTIL CHARNO < 0
1792  A7        MOVE20: AND    A                   ; REPEAT
1793  3D                DEC    A                   ;   LINENO :- 1
1794  ED5A              ADC    HL,DE               ;   CHARNO :+ 80
1796  FA9217            JP     M,MOVE20            ; UNTIL CHARNO >= 0
1799  320B01            LD     (LINENO),A
179C  7D                LD     A,L
179D  320A01            LD     (CHARNO),A
17A0  D1                POP    DE
17A1  2A0801            LD     HL,(CURSOR)         ; CURSOR :+ HL
17A4  19                ADD    HL,DE
17A5  220801            LD     (CURSOR),HL
17A8  C34F17            JP     GOTOXY              ; OUTCURSOR

                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                ;
                ; PROCEDURE PLACECURSOR
                ;
                ; INPUT : A : X-COORDINATE
                ;         B : Y-COORDINATE
                ;
                ; NO OUTPUT
                ;
                ; FUNCTION:
                ;            THE CURSOR IS MOVED TO THE INDICATED POSITION AND
                ;         THE 'LASTWASPRINTABLE' FLAG IS RESET.
                ;
                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

17AB            XPLACECURSOR:
17AB  320A01            LD     (CHARNO),A          ; CHARNO := A
17AE  6F                LD     L,A
17AF  2600              LD     H,0
17B1  78                LD     A,B
17B2  320B01            LD     (LINENO),A          ; LINENO := B
17B5  3AD315            LD     A,(#CHRLIN)         ; CURSOR := CHARNO +
17B8  5F                LD     E,A                 ;   LINENO*#CHRLIN
17B9  1600              LD     D,0
17BB  78                LD     A,B
17BC  B7                OR     A
17BD  2803              JR     Z,PLAC10
17BF  19        PLAC05: ADD    HL,DE
17C0  10FD              DJNZ   PLAC05
17C2            PLAC10
17C2  220801            LD     (CURSOR),HL
17C5  AF                XOR    A                   ; LASTWASPRINTABLE :=
17C6  320C01            LD     (LASTWASPRINTABLE),A; FALSE
17C9  C34F17            JP     GOTOXY              ; OUTCURSOR
```

```
                    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                    ;
                    ;  PROCEDURE BDOS
                    ;
                    ; STORES ALTERNATIVE REGISTER SET, IX AND IY
                    ; THE NECESSARY MAIN REGISTERS ARE STORED INSIDE
                    ; COMAL-80
                    ;
                    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
17CC  D9            BDOS:   EXX
17CD  E5                    PUSH    HL
17CE  D5                    PUSH    DE
17CF  C5                    PUSH    BC
17D0  DDE5                  PUSH    IX
17D2  FDE5                  PUSH    IY
17D4  D9                    EXX
17D5  CD0500                CALL    XBDOS
17D8  D9                    EXX
17D9  FDE1                  POP     IY
17DB  DDE1                  POP     IX
17DD  C1                    POP     BC
17DE  D1                    POP     DE
17DF  E1                    POP     HL
17E0  D9                    EXX
17E1  C9                    RET

17E2                        DEFS    100            ; SPACE FOR YOUR OWN
                                                   ; DRIVER.
                                                   ; USE THIS AREA FROM THE
                                                   ; LOWEST ADRESS UP, AS
                                                   ; PATCHES, IF IT BECOMES
                                                   ; NESSESARY, WILL USE THIS
                                                   ; AREA FROM THE TOP DOWN.
1846  00                    DEFB    0              ; BYTE SO THE ASSEMBLER
                                                   ; WORKS PROPERLY.
```

APPENDIX C
LIST OF ERROR MESSAGES

ERROR	TEXT
1	No more storage
2	Syntax error
3	Overflow
4	No $/# here
5	For strings only
6	Error in command
7	No more new names
8	String not terminated
9	Illegal character
10	Illegal character
11	Illegal line number
12	Line too long
13	Variable expected
14	')' expected
15	Type conflict
16	Expression too complicated
17	'(' expected
18	Type conflict in parameter
19	Has no parameters
20	Wrong type
21	',' expected
22	TAB not allowed here
23	Operand expected
24	Constant expected
25	':' expected
26	Function not allowed here
27	Illegal use of :=/:+/:-/=
28	:=/:+/:- expected
29	';' not allowed here
30	'FILE' expected
31	End-of-line here ?
32	Unknown device
33	A name expected
34	See manual
35	'OF' expected
36	Not a string function
37	Line number expected
38	GOTO/GOSUB expected
39	Illegal after 'THEN'
40	See manual
41	Array not allowed

42	TO/DOWNTO expected
43	READ/WRITE/RANDOM expected
44	From)= To
45	End-of-line expected
46	Statement expected
47	Command expected
48	Error in program structure
49	Type conflict
50	Error in program structure
51	Multiply defined
52	Function name expected
53	Name conflict with PROC/DEF
54	FOR-NEXT nesting depth
55	Unknown line number
56	RESTORE: to a data-statement only
57	Control structure not closed
58	Control structure not closed
59	Control structure not closed
60	Control structure not closed
61	Control structure not closed
62	Control structure not closed
63	Control structure not closed
64	Unknown PROC/DEF/LABEL
65	Program structure too complicated
66	'OUTPUT' expected
67	Index error
68	Illegal record number
69	No substrings here
70	Too few indices
71	Too many indices
72	Out of data
73	Error in assignment to substring
74	For arrays only

75	Error in the USING-string
76	Illegal TAB-value
77	Variable already exists
78	Cannot return
79	Name conflict with PROC/DEF
80	CASE-value not existing
81	STEP = 0
82	SYSTEM ERROR
83	SYSTEM ERROR
84	Out of domain
85	Too long
86	OVERFLOW
87	Undefined variable or function value
88	Too long
89	Not now
90	Index error
91	Type conflict in parameter
92	Too many parameters
93	Too few parameters
94	Division by 0
95	SYSTEM ERROR
96	Type conflict
97	Line too long
98	Not now
99	Error in NEXT
100	':' not allowed here
101	No line has such a number
102	Impossible
103	Impossible
104	Impossible
105	Auto overflow
106	!
107	Saved under an incompatible COMAL-version
108	Arrays must carry REF
109	The parameter must be a variable
110	The parameter has a wrong dimension
111	EXIT without LOOP
112	Control structure not closed

113	The channel is already open
114	The channel is not open
115	Illegal channel number
116	Unknown i/o device
117	Unknown i/o device
118	Error in filename
119	Error in filetype
120	Error in version number
121	No filetype stated
122	Filetype not allowed here
123	SYSTEM ERROR
124	SYSTEM ERROR
125	SYSTEM ERROR
126	Cannot write
127	Cannot read
128	Already open in another mode
129	File in use
130	SYSTEM ERROR
131	Cannot open more disk files
132	Non-existing file
133	Version number not allowed here
134	SYSTEM ERROR
135	SYSTEM ERROR
136	Impossible as a file is open
137	SYSTEM ERROR
138	Simple i/o device
139	SYSTEM ERROR
140	SYSTEM ERROR
141	SYSTEM ERROR
142	File catalog full
143	Disk or file full
144	SYSTEM ERROR
145	Illegal use of the file
146	"End-of-file"
147	SYSTEM ERROR
148	SYSTEM ERROR
149	Wrong block length
150	Control structure not closed
151	The channel is already open
152	The channel is not open

153 Illegal channel number
154 Unknown i/o device
155 Unknown i/o device
156 Error in filename
157 Error in filetype
158 Error in version number
159 No filetype stated
160 Filetype not allowed here
161 SYSTEM ERROR
162 SYSTEM ERROR
163 SYSTEM ERROR
164 Cannot write
165 Cannot read
166 Already open in another mode
167 File in use
168 SYSTEM ERROR
169 Cannot open more disk files
170 Non-existing file
171 Version number not allowed here
172 SYSTEM ERROR
173 SYSTEM ERROR
174 Impossible as a file is open
175 SYSTEM ERROR
176 Simple i/o device
177 SYSTEM ERROR
178 SYSTEM ERROR
179 SYSTEM ERROR
180 File catalog full
181 Disk or file full
182 SYSTEM ERROR
183 Illegal use of the file
184 "End-of-file"
185 SYSTEM ERROR
186 SYSTEM ERROR
187 Wrong block length
188 SYSTEM ERROR
189 SYSTEM ERROR
190 SYSTEM ERROR
191 SYSTEM ERROR
192 SYSTEM ERROR
193 SYSTEM ERROR
194 SYSTEM ERROR
195 SYSTEM ERROR

196	SYSTEM ERROR
197	SYSTEM ERROR
198	SYSTEM ERROR
199	SYSTEM ERROR
200	Control structure not closed
201	The channel is already open
202	The channel is not open
203	Illegal channel number
204	Unknown i/o device
205	Unknown i/o device
206	Error in filename
207	Error in filetype
208	Error in version number
209	No filetype stated
210	Filetype not allowed here
211	SYSTEM ERROR
212	SYSTEM ERROR
213	SYSTEM ERROR
214	Cannot write
215	Cannot read
216	Already open in another mode
217	File in use
218	SYSTEM ERROR
219	Cannot open more disk files
220	Non-existing file
221	Version number not allowed here
222	SYSTEM ERROR
223	SYSTEM ERROR
224	Impossible as a file is open
225	SYSTEM ERROR
226	Simple i/o device
227	SYSTEM ERROR
228	SYSTEM ERROR
229	SYSTEM ERROR
230	File catalog full
231	Disk or file full
232	SYSTEM ERROR
234	Illegal use of the file
235	"End-of-file"
236	SYSTEM ERROR
237	SYSTEM ERROR

238 Wrong block length
239 SYSTEM ERROR
240 SYSTEM ERROR
241 SYSTEM ERROR
242 SYSTEM ERROR
243 SYSTEM ERROR
244 SYSTEM ERROR
245 SYSTEM ERROR
246 SYSTEM ERROR
247 SYSTEM ERROR
248 SYSTEM ERROR
249 SYSTEM ERROR
250 SYSTEM ERROR
251 SYSTEM ERROR
252 SYSTEM ERROR
253 SYSTEM ERROR
254 SYSTEM ERROR
255 SYSTEM ERROR
256 SYSTEM ERROR
257 SYSTEM ERROR
258 Record exceeded
259 Illegal record length
260 This is not a RANDOM file
261 Wrong record length
262 Existing file
263 Impossible
264 Version number not
allowed here
265 Error in filename
266 Different i/o devices specified
267 SYSTEM ERROR
268 SYSTEM ERROR
269 SYSTEM ERROR
270 SYSTEM ERROR
271 SYSTEM ERROR
272 SYSTEM ERROR
273 SYSTEM ERROR
274 SYSTEM ERROR
275 SYSTEM ERROR
276 SYSTEM ERROR
277 SYSTEM ERROR
278 SYSTEM ERROR
279 SYSTEM ERROR
280 SYSTEM ERROR
281 SYSTEM ERROR
282 SYSTEM ERROR
283 SYSTEM ERROR
284 SYSTEM ERROR

285 SYSTEM ERROR
286 SYSTEM ERROR
287 SYSTEM ERROR
288 SYSTEM ERROR
289 SYSTEM ERROR
290 SYSTEM ERROR
291 SYSTEM ERROR
292 SYSTEM ERROR
293 SYSTEM ERROR

DEMONSTRATION PROGRAMS

```
0010 // PRIME FACTORING PROGRAM
0020 //
0030 // ASK FOR A NUMBER AND TEST IT
0040 //
0050 LOOP
0060   INPUT "INPUT POSITIVE INTEGER TO BE FACTORED: ": NUMBER
0070   IF NUMBER>0 AND FRAC(NUMBER)=0 THEN EXIT //TEST FOR POSITIVE
0080   //                                          INTEGER
0090   PRINT "I ASKED FOR A POSITIVE INTEGER!"
0100 ENDLOOP
0110 PRINT "THE PRIME FACTORS ARE: "
0120 //
0130 // PRIME 2 AND 3 MUST BE TREATED SEPARATELY
0140 //
0150 DIVISOR:=2
0160 EXEC TEST
0170 DIVISOR:=3
0180 EXEC TEST
0190 //
0200 //ALL PRIMES CAN BE EXPRESSED AS
0210 //N*6+5 AND N*6+7
0220 //
0230 FOR N:=0 TO SQR(NUMBER)/6 DO
0240   DIVISOR:=6*N+5
0250   EXEC TEST
0260   DIVISOR:=6*N+7
0270   EXEC TEST
0280 NEXT N
0290 IF NUMBER<>1 THEN PRINT NUMBER
0300 //
0310 PROC TEST
0320   WHILE NUMBER MOD DIVISOR=0 DO
0330     PRINT DIVISOR;
0340     NUMBER:=NUMBER DIV DIVISOR
0350   ENDWHILE
0360 ENDPROC TEST
```

```
0010 // CHARACTER SORT PROGRAM
0020 DIM STRING$ OF 2000
0030 DIM CHARACTER$ OF 1
0040 DIM COUNTER(ORD("A"):ORD("Z"))
0050 SPECIAL_CHARACTERS:=0
0060 SPACES:=0
0070 TRAP ESC- // TAKE CARE. SAVE THE PROGRAM
0080 //
0090 PRINT "INPUT A STRING: ",
0100 LOOP
0110   EXEC GET_CHARACTER(CHARACTER$) // GET CHARACTERS ONE BY ONE
0120   IF CHARACTER$=""27"" THEN EXIT
0130   PRINT CHARACTER$,
0140   STRING$:+CHARACTER$ // CONCATENATE CHARACTERS
0150 ENDLOOP // "ESC" TERMINATES INPUT
0160 PRINT
0170 //
0180 FOR I:=1 TO LEN(STRING$) DO
0190   CHARACTER$:=STRING$(I)
0200   IF CHARACTER$=" " THEN SPACES:+1 // TEST FOR SPACE
0210   IF CHARACTER$)="A" AND CHARACTER$(="Z" THEN // LETTER?
0220     COUNTER(ORD(CHARACTER$)):+1 // COUNT LETTER
0230   ELSE
0240     SPECIAL_CHARACTERS:+1 // COUNT OTHER CHARACTERS
0250   ENDIF
0260 NEXT I // GET NEXT CHARACTER
0270 // SET UP THE PRINT OUT FORMAT
0280 FOR J:=ORD("A") TO ORD("Z") DO // PRINT THE LETTERS
0290   PRINT "  ",CHR$(J),
0300 NEXT J
0310 PRINT // EMPTY LINE
0320 FOR K:=ORD("A") TO ORD("Z") DO // PRINT THE COUNT
0330   PRINT USING " ##": COUNTER(K),
0340 NEXT K
0350 PRINT
0360 PRINT
0370 PRINT "NUMBER OF CHARACTERS: ",LEN(STRING$)
0380 PRINT
0390 PRINT "NUMBER OF SPECIAL CHARACTERS INCLUDING SPACES: ",
0400 PRINT SPECIAL_CHARACTERS
0410 PRINT
0420 PRINT "NUMBER OF SPECIAL CHARACTERS EXCLUDING SPACES: ",
0430 PRINT SPECIAL_CHARACTERS-SPACES
0440 PROC GET_CHARACTER(REF A$) // LIBRARY PROCEDURE
0450   POKE 256, 255
0460   REPEAT
0470     IF ESC THEN POKE 256, 27
0480   UNTIL PEEK(256)()255
0490   A$:=CHR$(PEEK(256))
0500 ENDPROC GET_CHARACTER
```

```
0010 // CHANGING BASES
0020 // THIS PROGRAM WILL CHANGE A POSITIVE INTEGER BASE 10
0030 // TO ANY NEW BASE BETWEEN 2 AND 16
0040 DIM VALUE$(0:15) OF 1
0050 DIM DIGIT(20)
0060 FOR I:=0 TO 15 DO
0070   //
0080   // SET UP THE CHARACTER SET USED FOR OUTPUT
0090   //
0100   READ VALUE$(I)
0110 NEXT I
0120 DATA "0", "1", "2", "3", "4", "5", "6", "7"
0130 DATA "8", "9", "A", "B", "C", "D", "E", "F"
0140 //
0150 // GET THE NEW BASE AND TEST IT
0160 //
0170 REPEAT
0180   INPUT "NEW BASE: ": NEW_BASE
0190 UNTIL 2(=NEW_BASE AND NEW_BASE(=16 AND FRAC(NEW_BASE)=0
0200 //
0210 // GET THE NUMBER TO CONVERT
0220 //
0230 REPEAT
0240   INPUT "POSITIVE INTEGER TO BE CONVERTED: ": VALUE
0250   V:=VALUE
0260 UNTIL FRAC(VALUE)=0 AND VALUE)0
0270 //
0280 // CONVERT
0290 //
0300 I:=1
0310 REPEAT
0320   DIGIT(I):=VALUE MOD NEW_BASE; VALUE:=VALUE DIV NEW_BASE
0330   I:+1
0340 UNTIL VALUE=0
0350 NO_DIGITS:=I-1
0360 //
0370 // PRINT THE RESULT
0380 //
0390 PRINT VALUE," BASE 10 CONVERTS IN BASE ",NEW_BASE," TO: ",
0400 FOR I:=NO_DIGITS DOWNTO 1 DO
0410   PRINT VALUE$(DIGIT(I))," ",
0420 NEXT I
```

```
0010 //  LISSAJOUS PATTERNS
0020 //
0030 // CONSTANTS DEFINING THE SCREEN.
0040 // HALVE THE VALUES FOR 40-CHARACTER SCREENS.
0050 // ADJUST 'SCALE' TO YOUR SCREEN SO THAT INPUTS 1, 1 AND 0.5
0060 // PRODUCE A PERFECT CIRCLE.
0070 //
0080 SCALE:=27
0090 CHARACTERS:=80 // NUMBER OF CHARACTERS ACROSS THE SCREEN
0100 LINES:=24 // NUMBER OF LINES ON THE SCREEN
0110 //
0120 ADJUST:=INT((CHARACTERS-2*SCALE-1)/2)
0130 IF ADJUST<0 THEN STOP
0140 X_LIMIT:=(LINES-2)/2
0150 //
0160 DIM LINE$ OF CHARACTERS
0170 PI:=3.14159
0180 CLEAR
0190 //
0200 REPEAT
0210   INPUT "RELATIVE FREQ. FOR X: ": X_REL_FREQ // TRY 4
0220 UNTIL FRAC(X_REL_FREQ)=0 AND X_REL_FREQ)=1
0230 NO_STEPS:=X_REL_FREQ; X_REL_FREQ:=2*PI*X_REL_FREQ
0240 //
0250 REPEAT
0260   INPUT "RELATIVE FREQ. FOR Y: ": Y_REL_FREQ // TRY 3
0270 UNTIL FRAC(Y_REL_FREQ)=0 AND Y_REL_FREQ)=1
0280 Y_REL_FREQ:=2*PI*Y_REL_FREQ
0290 //
0300 INPUT "Y PHASE, MULTIPLE OF PI: ": Y_PHASE // TRY 0
0310 Y_PHASE:=PI*Y_PHASE
0320 //
0330 CLEAR
0340 FOR X_STEP:=X_LIMIT DOWNTO -X_LIMIT DO
0350   LINE$:=SPC$(CHARACTERS)
0360   X:=FN_ARCSIN(X_STEP/X_LIMIT)
0370   FOR I:=0 TO NO_STEPS-1 DO
0380     LINE$(FN_SCALED(X,I)):="*"
0390     LINE$(FN_SCALED(PI-X,I)):="*"
0400   NEXT I
0410   PRINT LINE$
0420 NEXT X_STEP
0430 CURSOR 1, LINES-1
0440 END
0450 //
```

```
0460 DEF FN_ARCSIN(X)
0470   IF ABS(X)<0.1 THEN
0480     FN_ARCSIN:=X+X^3/6+X^5*0.075+X^7/22.4
0490   ELSE
0500     FN_ARCSIN:=2*FN_ARCSIN(X/(SQR(1+X)+SQR(1-X)))
0510   ENDIF
0520 ENDDEF FN_ARCSIN
0530 //
0540 DEF FN_COMPUTE(T, I)
0550   GLOBAL PI, X_REL_FREQ, Y_REL_FREQ, Y_PHASE
0560   TT:=(T+2*I*PI)/X_REL_FREQ
0570   FN_COMPUTE:=SIN(Y_REL_FREQ*TT+Y_PHASE)
0580 ENDDEF FN_COMPUTE
0590 //
0600 DEF FN_SCALED(T, I)
0610   GLOBAL SCALE, ADJUST
0620   FN_SCALED:=1+ADJUST+ROUND(SCALE*(FN_COMPUTE(T,I)+1))
0630 ENDDEF FN_SCALED
```

```
0010 // WRITTEN october -81
0020 // by H.C. Grosbll-Poulsen, Gl.Rye, Denmark
0030 //
0040 // DESCRIPTION of the procedure 'EDITLINE'
0050 // The procedure is closed, qualifying it for
0060 // immediate inclusion in the User's library.
0070 // PURPOSE: to edit a textvariable written on
0080 // the screen, thus the procedure is effectively
0090 // a lineeditor.
0100 // PARAMETERS: ORG_X# and ORG_Y# are integers
0110 // (valueparameter) describing the coordinates
0120 // of the position where the textvariable
0130 // originally was written.
0140 // REF LINE$ is the textvariable. It is a variable-
0150 // parameter, so that the editing is refered back
0160 // to the invocating variable.
0170 // REF KEYBOARD# is an integer, whose sole purpose
0180 // is to refer back the last input from the
0190 // keyboard for further processing in the calling
0200 // program. Value by entrance is of no significance.
0210 //
0220 // Example:
0230 //          CURSOR 20, 15
0240 //          PRINT TEXT$(I);
0250 //          EXEC EDITLINE(20,15,TEXT$(I),A#)
0260 //
0270 //-------------------------------------------------------
0280 //
0290 PROC EDITLINE(ORG_X#, ORG_Y#, REF LINE$, REF KEYBOARD#) CLOSED
0300   DIM CODE$ OF 15, HELP$ OF 80 //   NB: The length may vary
0310   X#:=1; RETURNBACK:=FALSE
0320   EXEC INDATAINIT
0330   CURSOR ORG_X#, ORG_Y#
0340   REPEAT
0350     EXEC INDATA(KEYBOARD#,MACHINECODE)
0360     CASE KEYBOARD# OF
0370     WHEN 13, 11, 10 //     refer to ASCII-table
0380       RETURNBACK:=TRUE
0390     WHEN 8
0400       EXEC CURSORLEFT
0410     WHEN 12
0420       EXEC CURSORRIGHT
0430     WHEN 127
0440       EXEC DELETEBYTE
0450     WHEN 31
0460       EXEC INSERTBLANK
0470     OTHERWISE
0480       EXEC WRITEBYTE
0490     ENDCASE
0500   UNTIL RETURNBACK
0510 ENDPROC EDITLINE
```

```
0520 //
0530 //
0540 PROC CURSORLEFT //  if possible, move cursor left
0550   IF X#)1 THEN
0560     X#:-1
0570     CURSOR ORG_X#+X#-1, ORG_Y#
0580   ENDIF
0590 ENDPROC CURSORLEFT
0600 //
0610 //
0620 PROC CURSORRIGHT //  if possible, move right
0630   IF X#-1(LEN(LINE$) THEN
0640     X#:+1
0650     CURSOR ORG_X#+X#-1, ORG_Y#
0660   ENDIF
0670 ENDPROC CURSORRIGHT
0680 //
0690 //
0700 PROC INSERTBLANK //  test for extreme positioning
0710   IF LEN(LINE$))X#-1 THEN //          of the cursor
0720     HELP$:=LINE$(X#:LEN(LINE$))
0730   ELSE
0740     HELP$:=""
0750   ENDIF
0760   IF X#)1 THEN
0770     LINE$:=LINE$(1,X#-1)
0780   ELSE
0790     LINE$:=""
0800   ENDIF
0810   LINE$:+" "+HELP$
0820   EXEC REWRITELINE
0830 ENDPROC INSERTBLANK
0840 //
0850 //
0860 PROC LINETEST //  test for extreme positioning
0870   IF LEN(LINE$))X# THEN //         of the cursor
0880     HELP$:=LINE$(X#+1:LEN(LINE$))
0890   ELSE
0900     HELP$:=""
0910   ENDIF
0920   IF X#)1 THEN
0930     LINE$:=LINE$(1,X#-1)
0940   ELSE
0950     LINE$:=""
0960   ENDIF
0970 ENDPROC LINETEST
0980 //
0990 //
```

```
1000 PROC DELETEBYTE
1010   EXEC LINETEST
1020   LINE$:+HELP$
1030   EXEC REWRITELINE
1040 ENDPROC DELETEBYTE
1050 //
1060 //
1070 PROC WRITEBYTE
1080   EXEC LINETEST
1090   LINE$:+CHR$(KEYBOARD#)+HELP$
1100   EXEC REWRITELINE
1110   EXEC CURSORRIGHT
1120 ENDPROC WRITEBYTE
1130 //
1140 //
1150 PROC REWRITELINE //  used after writing, deletion
1160   CURSOR ORG_X#, ORG_Y# //       or insertion of a
1170   PRINT LINE$+" "; //                    character
1180   CURSOR ORG_X#+X#-1, ORG_Y#
1190 ENDPROC REWRITELINE
1200 //
1210 //
1220 PROC INDATAINIT //         place machinecode in the space
1230   MACHINECODE:=VARPTR(CODE$); B:=MACHINECODE // allocated
1240   POKE B, 30 // LD E,255                      for in CODE$
1250   POKE B+1, 255
1260   POKE B+2, 14 // LD C,6                  refer to Z80 and
1270   POKE B+3, 6
1280   POKE B+4, 205 // CALL BDOS                 CP/M manuals
1290   POKE B+5, 5
1300   POKE B+6, 0
1310   POKE B+7, 183 // OR A
1320   POKE B+8, 202 // JP NZ,B
1330   POKE B+9, B MOD 256
1340   POKE B+10, B DIV 256
1350   POKE B+11, 50 // LD (KEYBOARD#),A //   making the value
1360   POKE B+12, VARPTR(KEYBOARD#) MOD 256 //   accessible to
1370   POKE B+13, VARPTR(KEYBOARD#) DIV 256 //        COMAL-80
1380   POKE B+14, 210 // RET
1390 ENDPROC INDATAINIT
1400 //
1410 //
1420 PROC INDATA(REF KEYBOARD#, MACHINECODE) //        get an
1430   CALL MACHINECODE //         unechoed input from console
1440 ENDPROC INDATA
```

ASCII CHARACTER CODES

ASCII Code	CHARACTER	ASCII Code	CHARACTER	ASCII Code	CHARACTER
000	NUL	043	+	086	V
001	SOH	044	,	087	W
002	STX	045	-	088	X
003	ETX	046	.	089	Y
004	EOT	047	/	090	Z
005	ENQ	048	0	091	[
006	ACK	049	1	092	\
007	BEL	050	2	093	]
008	BS	051	3	094	^
009	HT	052	4	095	_
010	LF	053	5	096	`
011	VT	054	6	097	a
012	FF	055	7	098	b
013	CR	056	8	099	c
014	SO	057	9	100	d
015	SI	058	:	101	e
016	DLE	059	;	102	f
017	DC1	060	<	103	g
018	DC2	061	=	104	h
019	DC3	062	>	105	i
020	DC4	063	?	106	j
021	NAK	064	@	107	k
022	SYN	065	A	108	l
023	ETB	066	B	109	m
024	CAN	067	C	110	n
025	EM	068	D	111	o
026	SUB	069	E	112	p
027	ESC	070	F	113	q
028	FS	071	G	114	r
029	GS	072	H	115	s
030	RS	073	I	116	t
031	VS	074	J	117	u
032	SPACE	075	K	118	v
033	!	076	L	119	w
034	"	077	M	120	x
035	#	078	N	121	y
036	$	079	O	122	z
037	%	080	P	123	{
038	&	081	Q	124	|
039	'	082	R	125	}
040	(	083	S	126	
041	)	084	T	127	DEL
042	*	085	U		

ASCII codes are in decimal
LF=Line Feed, FF=Form Feed, CR=Carriage Return, DEL=Rubout

In our continuous efforts to improve this manual, METANIC ApS ask you, the user, to use this report and send us any correction, comment, suggestion, or addition that you may have to this manual.

The format of the COMAL-80 manual is designed for easy updating, and your report may well be included in the next update. Forwarded information becomes the property of METANIC ApS.

Please specify page and line references where applicable.

Manual Edition: _____

Errors: _____

Comments: _____

Name: _____ Date: _____

Address: _____

_____

_____

Country: _____

FORWARD TO: METANIC APS, KONGEVEJEN 177, DK-2830 VIRUM, DENMARK

METANIC COMAL-80

SYNTAX DIAGRAMS & EXAMPLES

Comal-80 EDP

-the key to programming

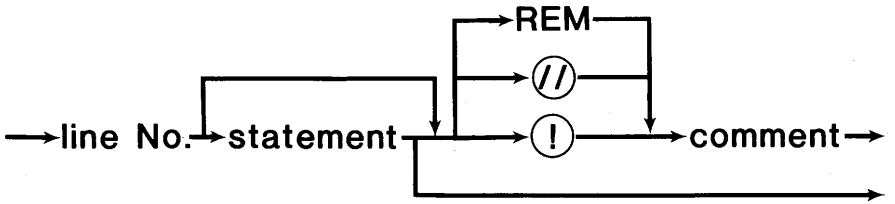
Acknowledgements:

METANIC hereby wishes to thank all the persons involved in specifying and testing of COMAL-80. A special acknowledgement is extended to Mr. Børge R. Christensen, DATO, Tønder.

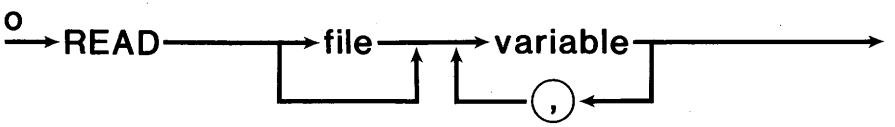
This booklet contains the total syntax diagrams for METANIC COMAL-80, Version 1.
Minor differences may occur in the implementation onto specific microcomputers. Please consult your manual for changes.

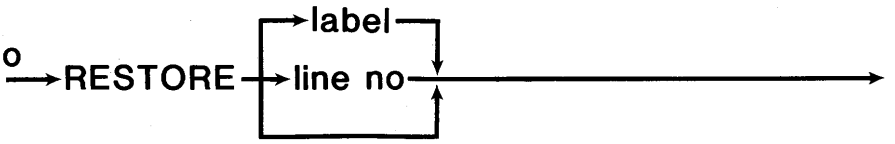
METANIC COMAL-80
SYNTAX DIAGRAMS
VERSION 1.

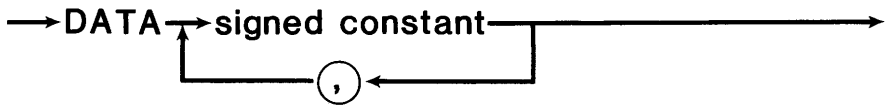
Line:

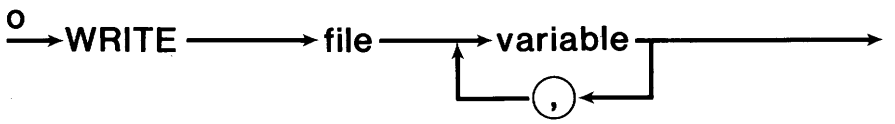


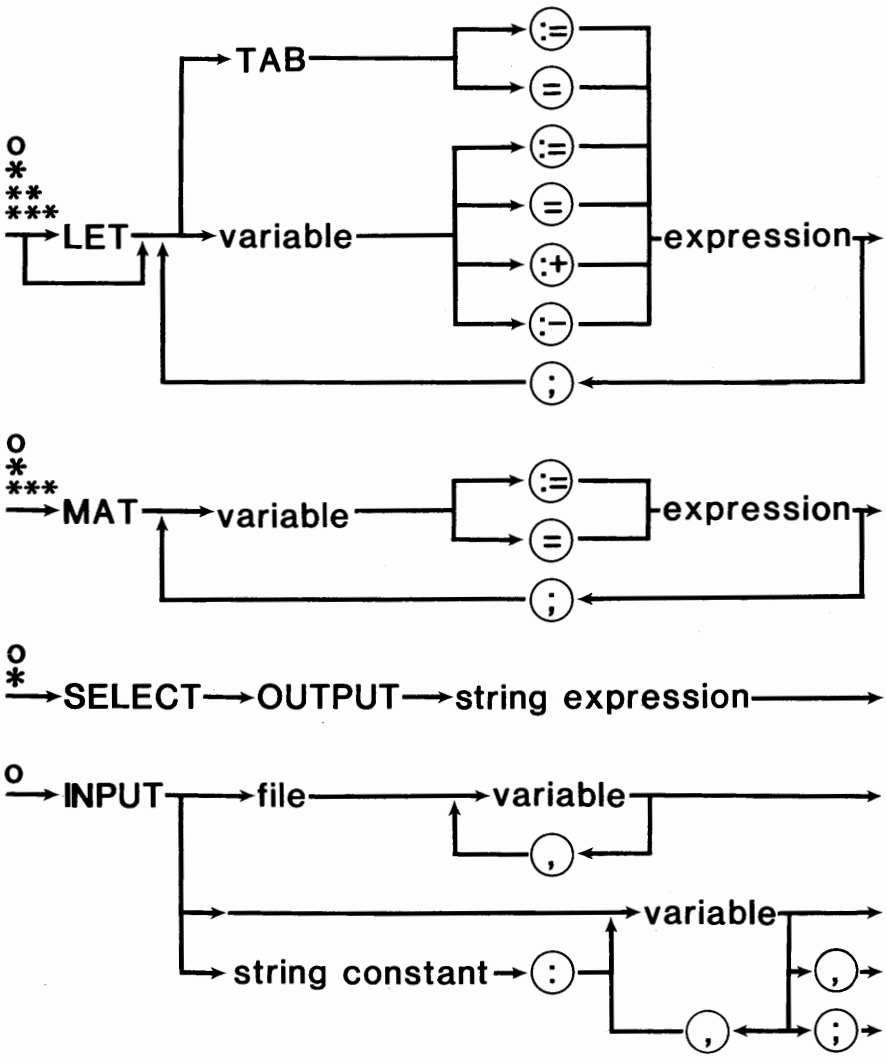
Statement:







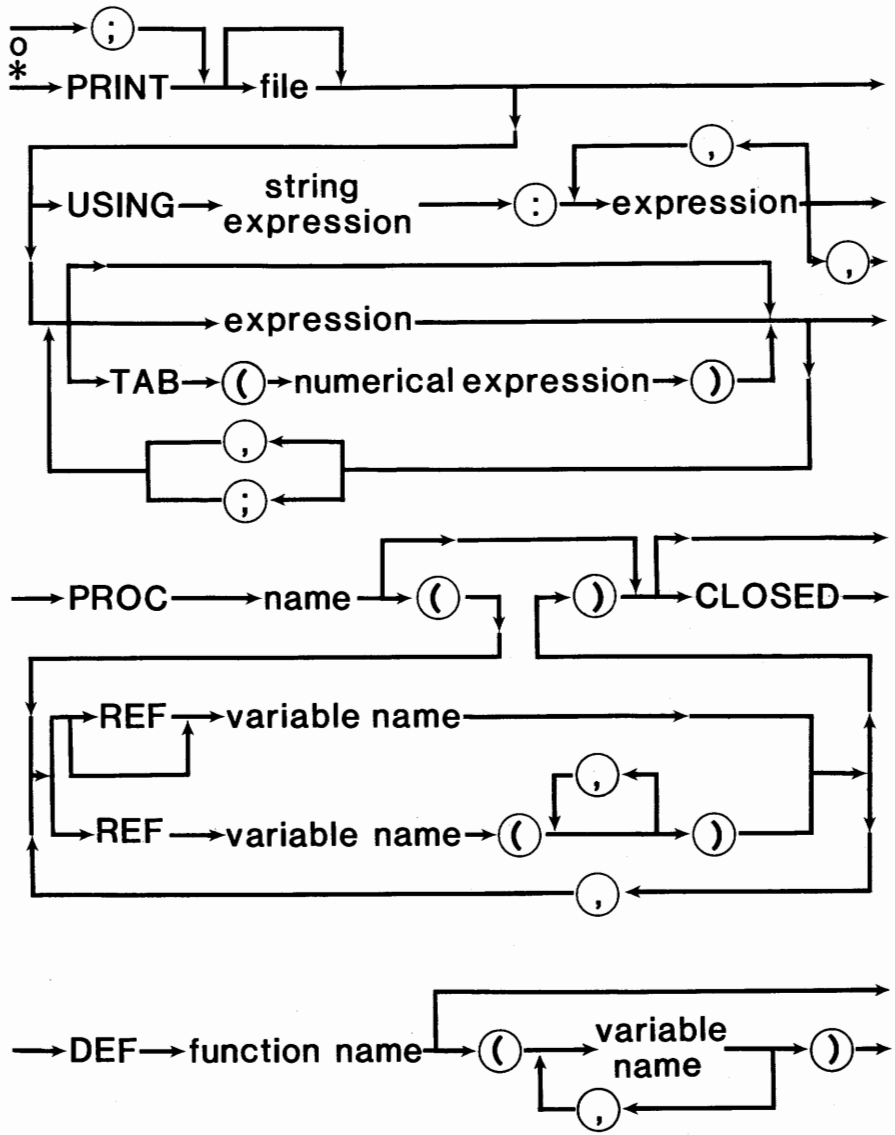


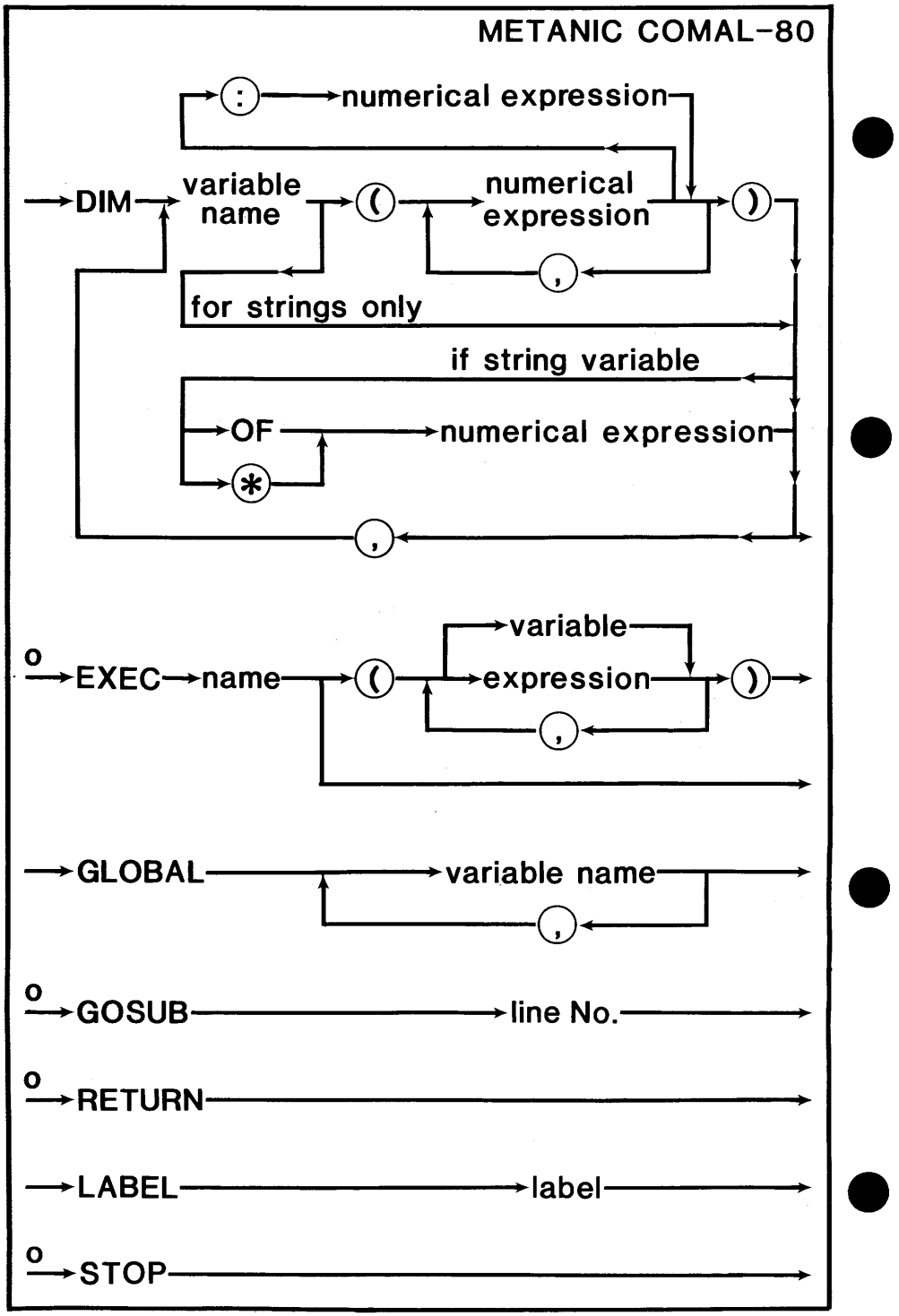


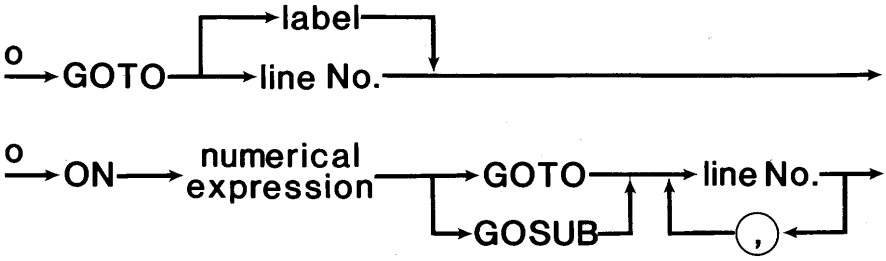
**
In connection with strings :- may not be used, whereas :+ may be used.

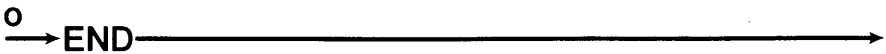
***
Variable and expression in one assignment must be of the same type. The only exception is:

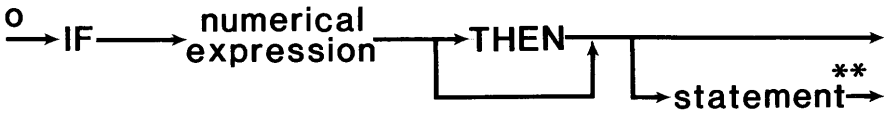
real variable := integer expression


o
*
;
PRINT
file
USING
string expression
:
,
expression
,
expression
TAB
(
numerical expression
)
,
;
PROC
name
(
)
CLOSED
REF
variable name
REF
variable name
(
,
)
,
DEF
function name
(
variable name
)
,
ENDPROC
name
ENDDEF
function name

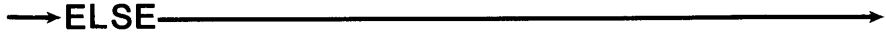

:
numerical expression
DIM
variable name
(
numerical expression
)
,
for strings only
if string variable
OF
numerical expression
*
,
o
EXEC
name
(
variable
expression
)
,
GLOBAL
variable name
,
o
GOSUB
line No.
o
RETURN
LABEL
label
o
STOP

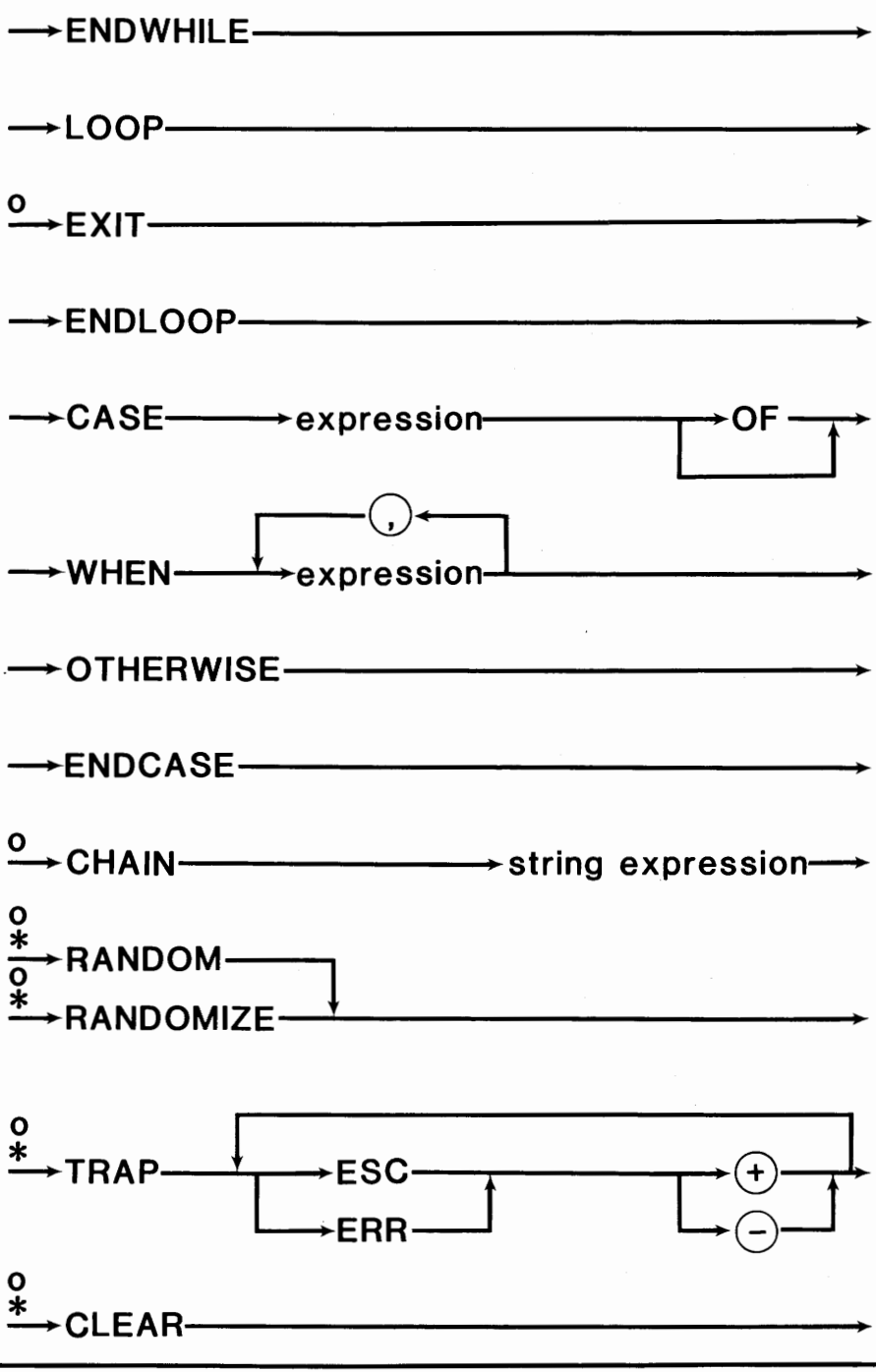


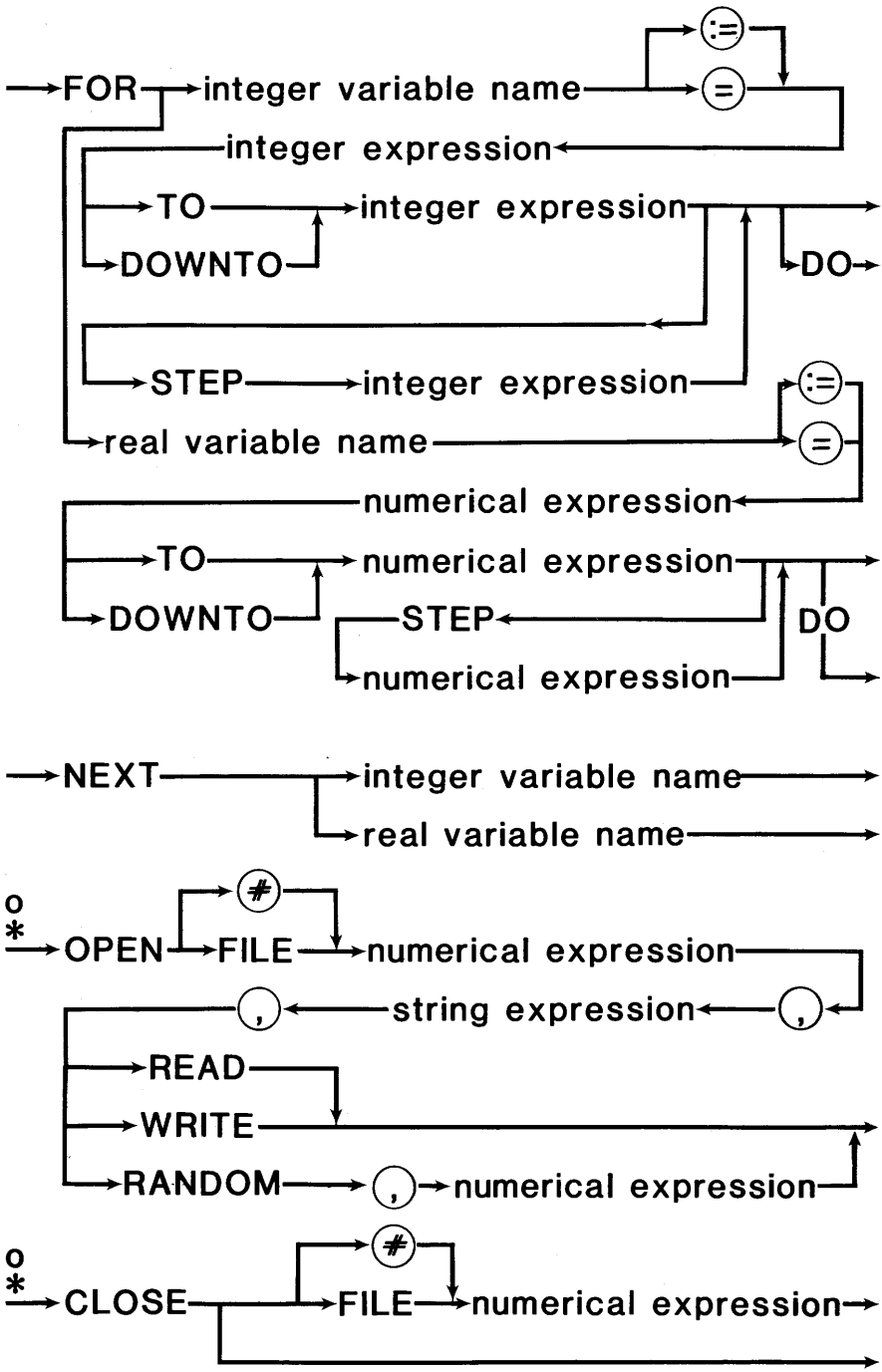


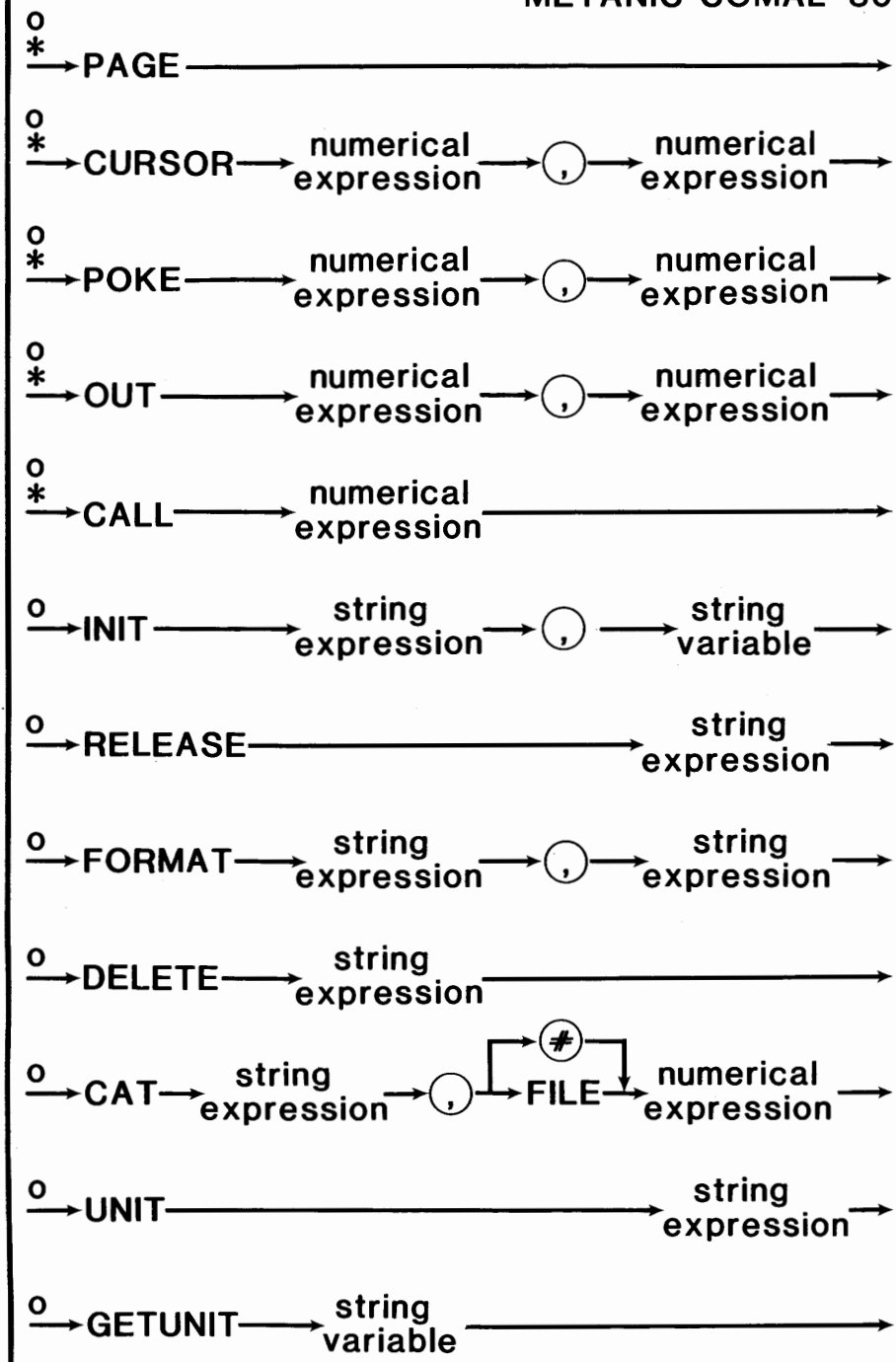


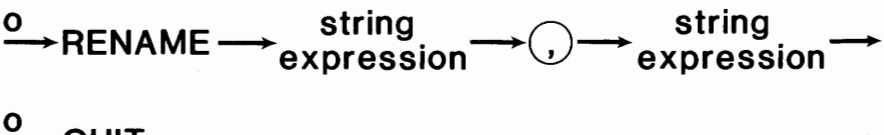
** Only statements marked o may be used here.



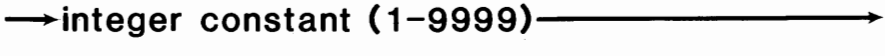

ENDWHILE
LOOP
EXIT
ENDLOOP
CASE
expression
OF
,
WHEN
expression
OTHERWISE
ENDCASE
CHAIN
string expression
RANDOM
RANDOMIZE
TRAP
ESC
ERR
+
−
CLEAR


FOR
integer variable name
:=
=
integer expression
TO
DOWNTO
integer expression
DO
STEP
integer expression
real variable name
:=
=
numerical expression
TO
DOWNTO
numerical expression
STEP
numerical expression
DO
NEXT
integer variable name
real variable name
o
*
OPEN
#
FILE
numerical expression
,
string expression
,
READ
WRITE
RANDOM
,
numerical expression
o
*
CLOSE
#
FILE
numerical expression

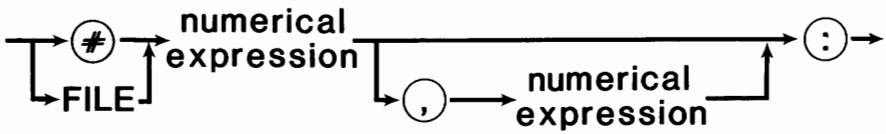

o
*
PAGE
o
*
CURSOR
numerical expression
,
numerical expression
o
*
POKE
numerical expression
,
numerical expression
o
*
OUT
numerical expression
,
numerical expression
o
*
CALL
numerical expression
o
INIT
string expression
,
string variable
o
RELEASE
string expression
o
FORMAT
string expression
,
string expression
o
DELETE
string expression
o
CAT
string expression
,
#
FILE
numerical expression
o
UNIT
string expression
o
GETUNIT
string variable



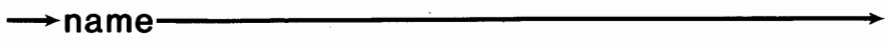
Line No.:



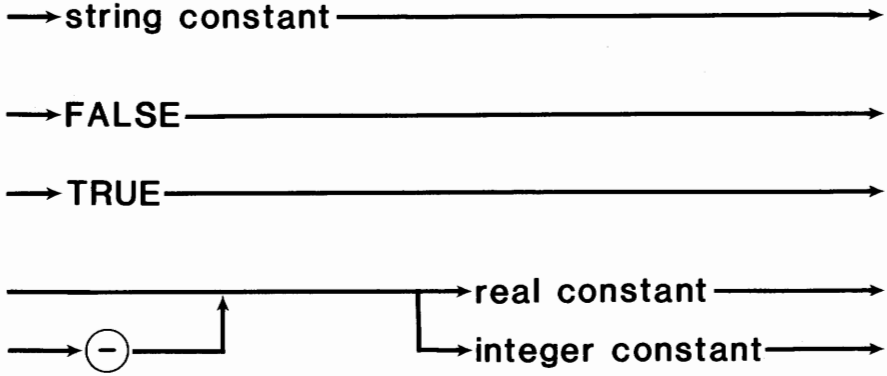
File:



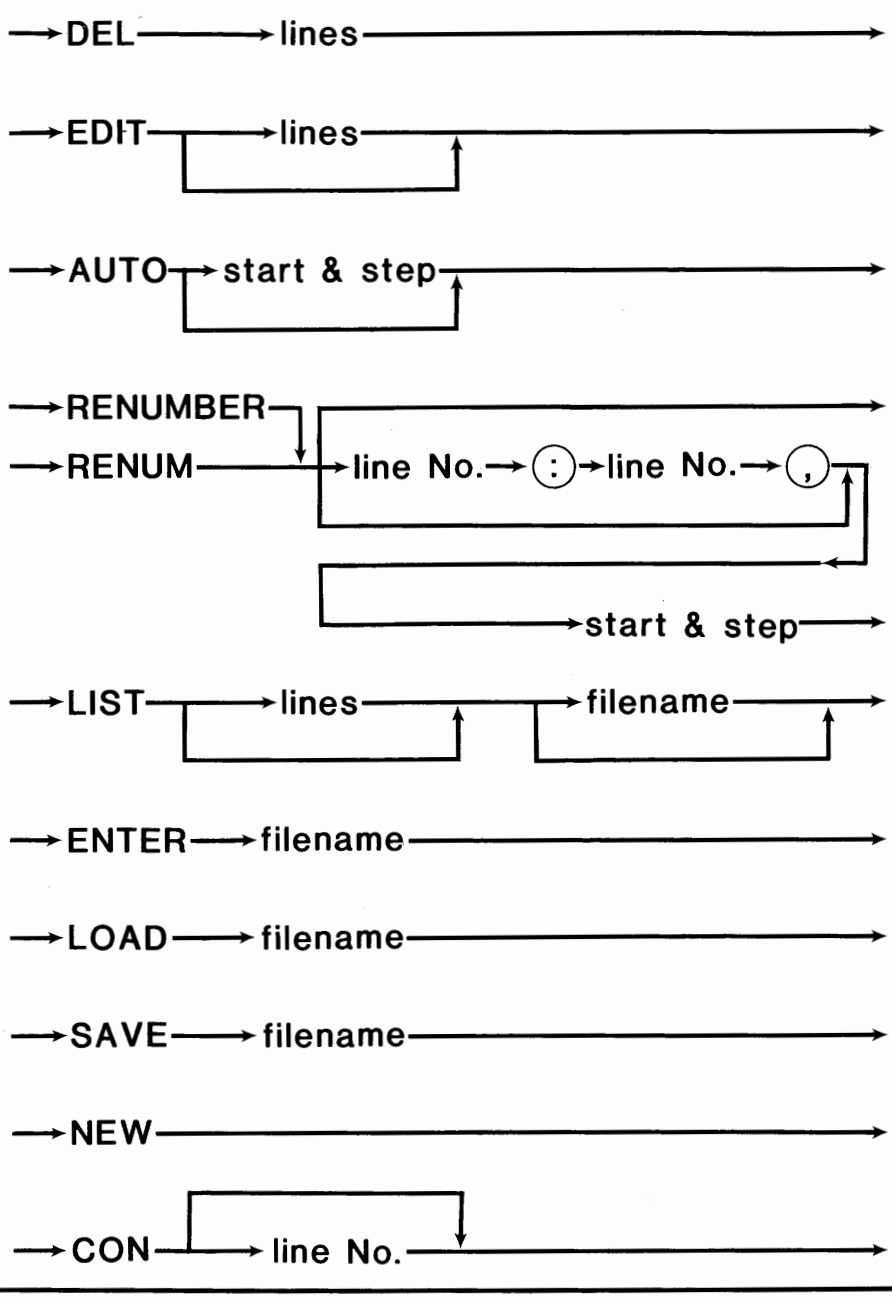
Label:

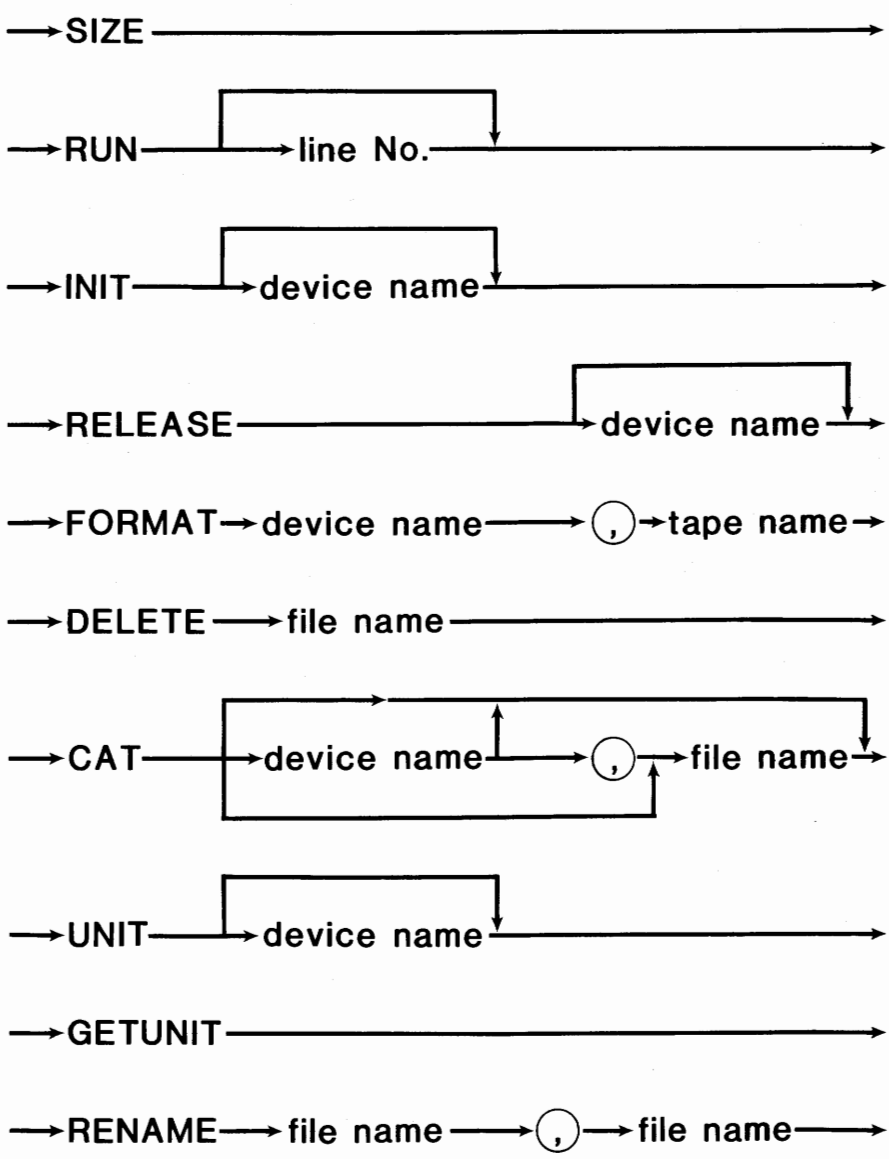


Signed Constant:



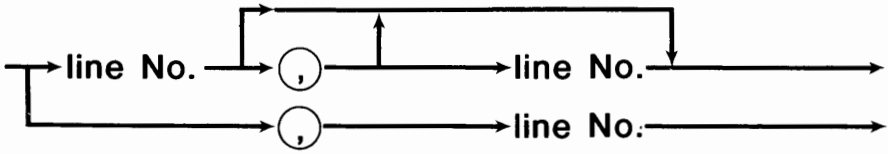
Command:


DEL
lines
EDIT
lines
AUTO
start & step
RENUMBER
RENUM
line No.
:
line No.
,
start & step
LIST
lines
filename
ENTER
filename
LOAD
filename
SAVE
filename
NEW
CON
line No.

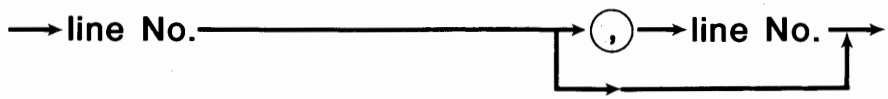


All statements marked * may be used as commands.

Lines:



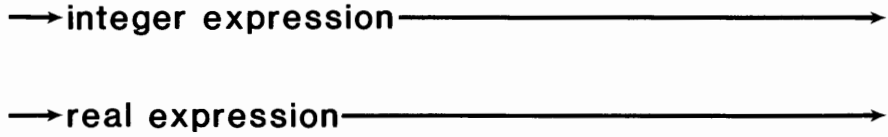
Start & Step:



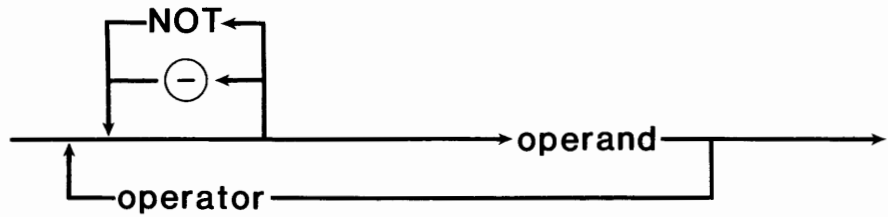
File Name & Device Name:

Any sequence of characters not starting with a digit, a comma, a space, or a colon, and not containing a comma or a space may be used.

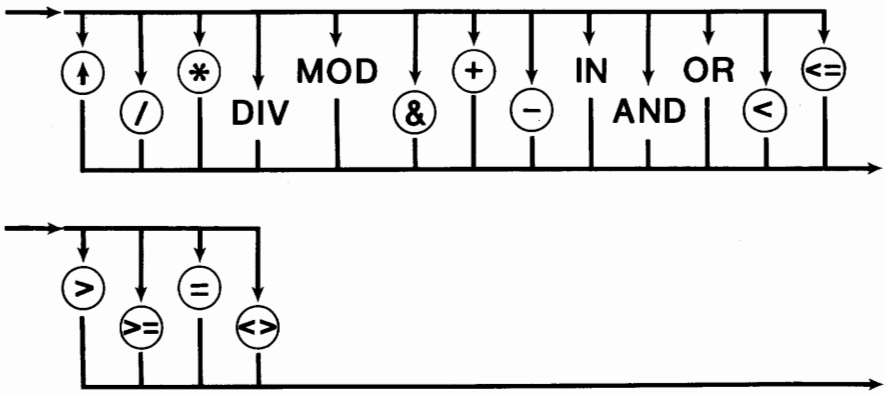
Numerical Expression:



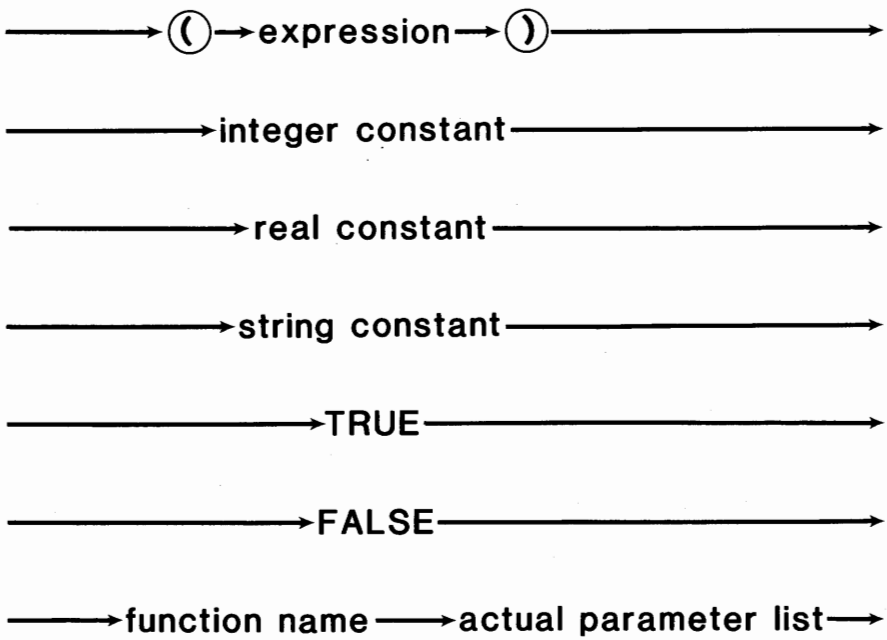
String-, Integer-,
& Real-Expressions:

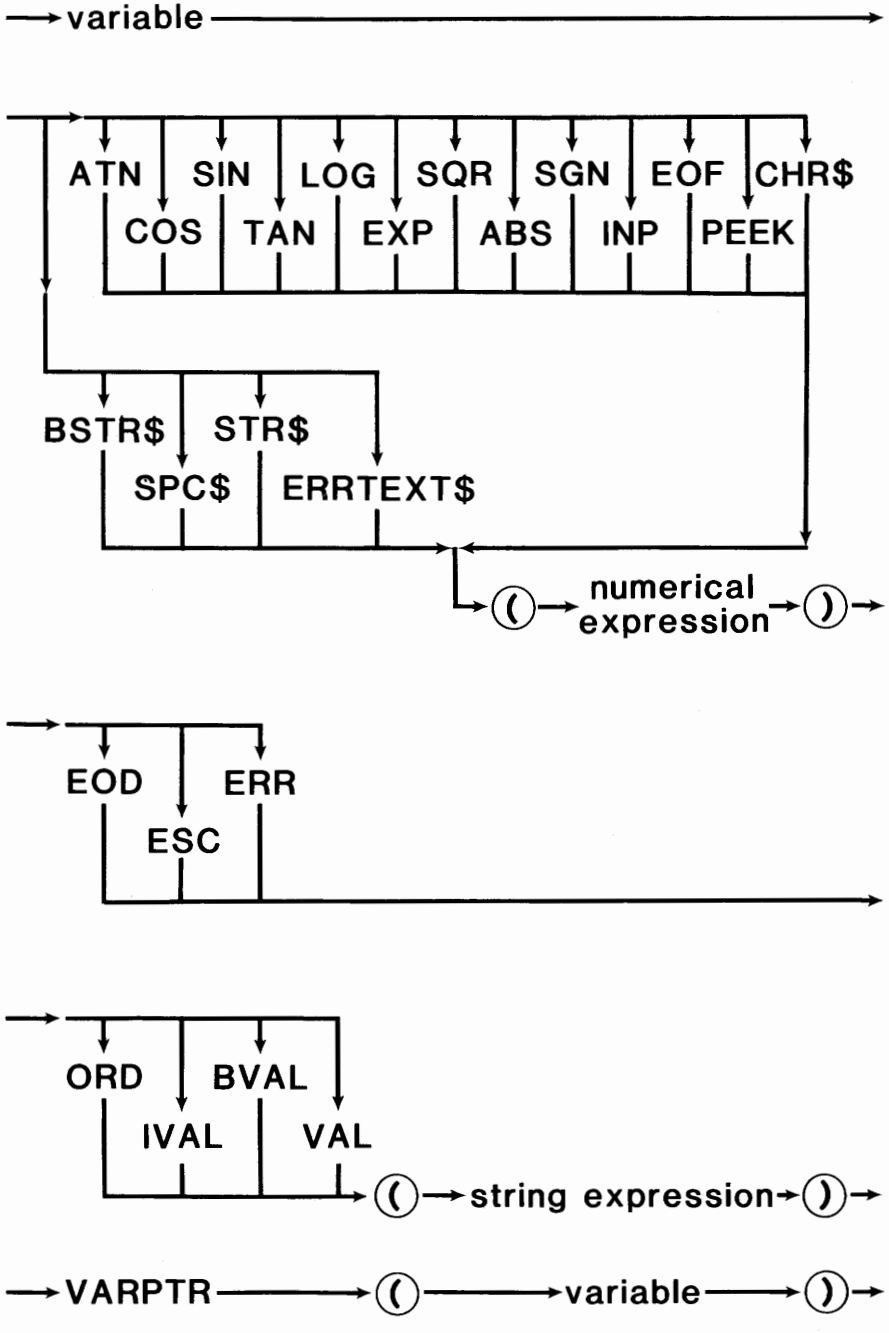


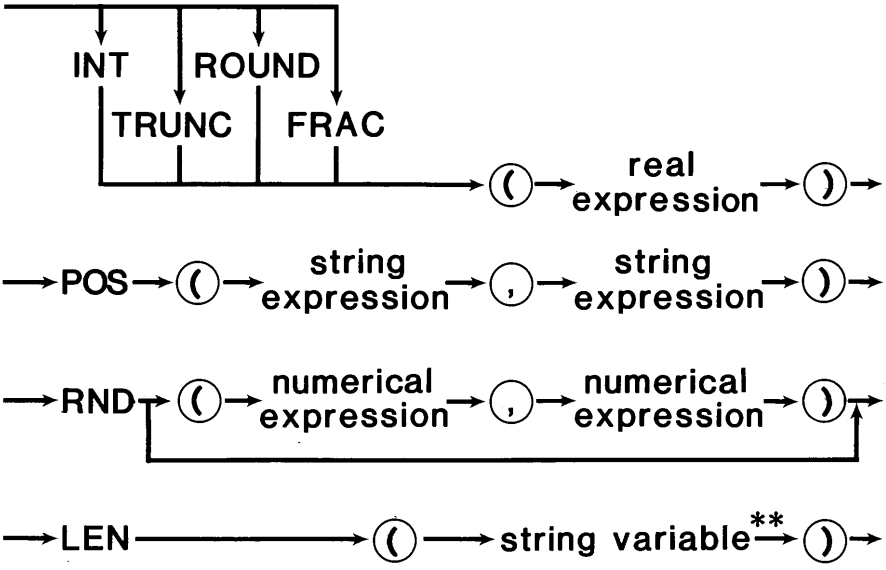
Operator:



Operand:

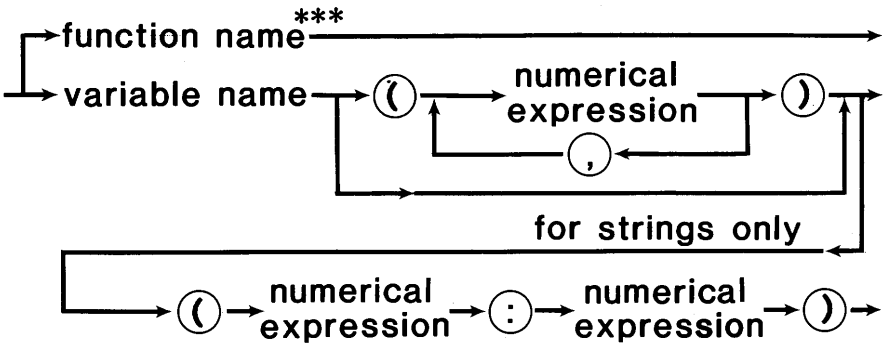



variable
ATN
SIN
LOG
SQR
SGN
EOF
CHR$
COS
TAN
EXP
ABS
INP
PEEK
BSTR$
STR$
SPC$
ERRTEXT$
(
numerical expression
)
EOD
ERR
ESC
ORD
BVAL
IVAL
VAL
(
string expression
)
VARPTR
(
variable
)



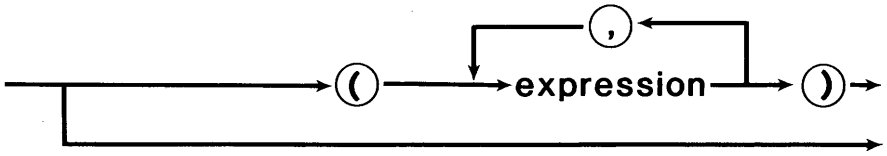
** Not substrings.

Variable:

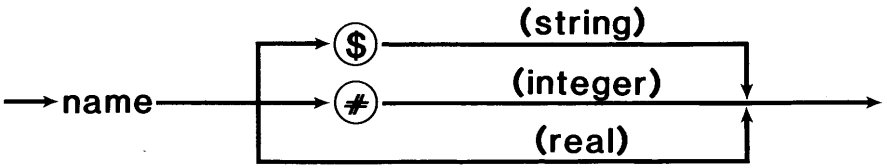


*** Can be substituted for variables in expressions and LET, READ, and INPUT statements only.

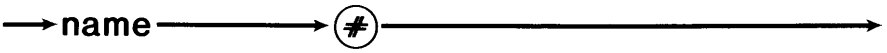
Actual Parameter List:



Variable name:

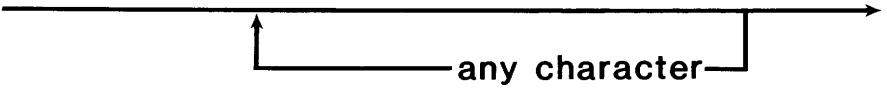


Integer Variable Name:

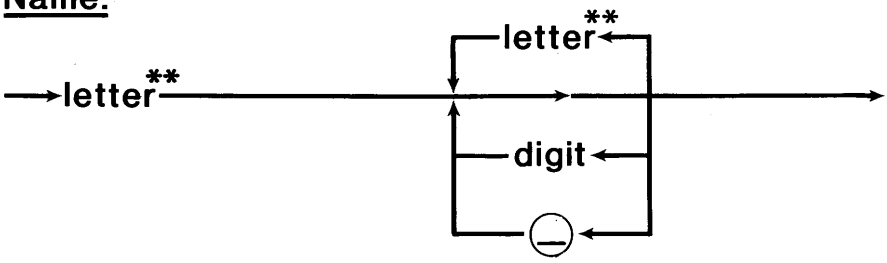


Real Variable Name:

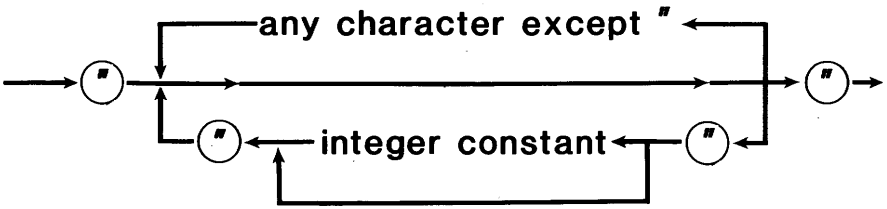
Comment & Tape Name:



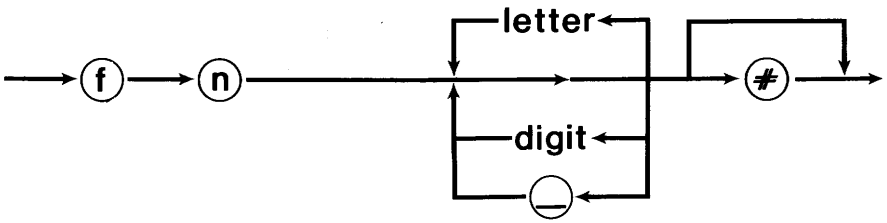
Name:



String Constant:



Function Name:



**
Names starting with fn are reserved for function names only.

```
# 1
0010 // ALL SOLUTIONS TO THE EIGHT-QUEENS
0020 // PROBLEM. FROM: ALGORITHMS + DATA
0030 // STRUCTURES = PROGRAMS BY N.WIRTH
0040 // BY ARNE CHRISTENSEN,1980
0050 //
0060 DIM A(1:8), B(2:16), C(-7:7), X(1:8)
0070 PROC PRINTING
0080   FOR K:=1 TO 8 DO
0090     PRINT USING "####": X(K),
0100   NEXT K
0110   PRINT
0120 ENDPROC PRINTING
0130 //
0140 PROC TRY(I) CLOSED
0150   GLOBAL A, B, C, X
0160   FOR J:=1 TO 8 DO
0170     IF A(J) AND B(I+J) AND C(I-J) THEN
0180       X(I):=J; A(J):=FALSE; B(I+J):=FALSE
0190       C(I-J):=FALSE
0200       IF I<8 THEN
0210         EXEC TRY(I+1)
0220       ELSE
0230         EXEC PRINTING
0240       ENDIF
0250       A(J):=TRUE; B(I+J):=TRUE; C(I-J):=TRUE
0260     ENDIF
0270   NEXT J
0280 ENDPROC TRY
0290 //
0300 MAT A:=TRUE; B:=TRUE; C:=TRUE
0310 EXEC TRY(1)
```

```
# 2
0010 // LABEL DEMONSTRATION
0020 // BY ARNE CHRISTENSEN, 1980
0030 LABEL AGAIN
0040 RESTORE DATA2
0050 READ X
0060 PRINT X
0070 RESTORE DATA1
0080 READ X
0090 PRINT X
0100 GOTO AGAIN
0110 LABEL DATA1
0120 DATA 47
0130 LABEL DATA2
0140 DATA -47

# 3
0010 SUM:=0
0020 FOR FIGURE#:=500 DOWNTO 1
0030  SUM:+ FIGURE#
0040 NEXT FIGURE#
0050 PRINT SUM

# 4
0010 DIM FIRST_NAME$ OF 10
0020 DIM FAMILY_NAME$ OF 10
0030 DATA "John", "Doe", 10
0040 READ FIRST_NAME$, FAMILY_NAME$
0050 PRINT FIRST_NAME$+" "+FAMILY_NAME$
0060 READ AGE
0070 PRINT AGE;  "YEAR"
```

```
# 5
0010 // LOOP AND CASE DEMONSTRATION
0020 // A SMALL RPN CALCULATOR PROGRAM
0030 // BY ARNE CHRISTENSEN, 1980
0040 DIM S(10), COMMAND$ OF 10
0050 MAT S:=0 // S IS THE STACK
0060 TOP:=0
0070 CLEAR // CLEAR SCREEN
0080 LOOP
0090  // PRINT OUT THE STACK
0100  CURSOR 1, 1 // UPPER LEFT
0110  FOR I:=1 TO TOP DO
0120   PRINT S(I);SPC$(20)
0130  NEXT I
0140  PRINT SPC$(20)
0150  // GET NEXT COMMAND
0160  CURSOR 1, TOP+3
0170  INPUT COMMAND$
0180  CURSOR 1, TOP+3
0190  PRINT SPC$(20)
0200  // EXECUTE COMMAND
0210  CASE COMMAND$ OF
0220  WHEN "+"
0230   TOP:-1; S(TOP):+S(TOP+1)
0240  WHEN "-"
0250   TOP:-1; S(TOP):-S(TOP+1)
0260  WHEN "*"
0270   TOP:-1; S(TOP):=S(TOP)*S(TOP+1)
0280  WHEN "/"
0290   TOP:-1; S(TOP):=S(TOP)/S(TOP+1)
0300  OTHERWISE
0310   TOP:+1; S(TOP):=VAL(COMMAND$)
0320  ENDCASE
0330 ENDLOOP
```

All statements marked * may be used as commands.

Only statements marked o may be used after IF....THEN.

Distributor:

Tryk: Kiwi Multiprint ApS · Gl. Rye