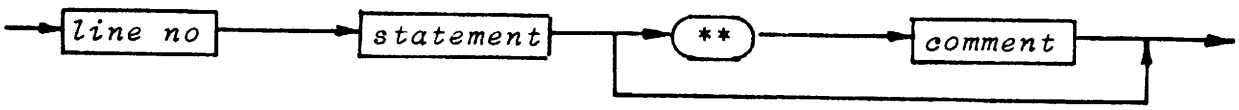
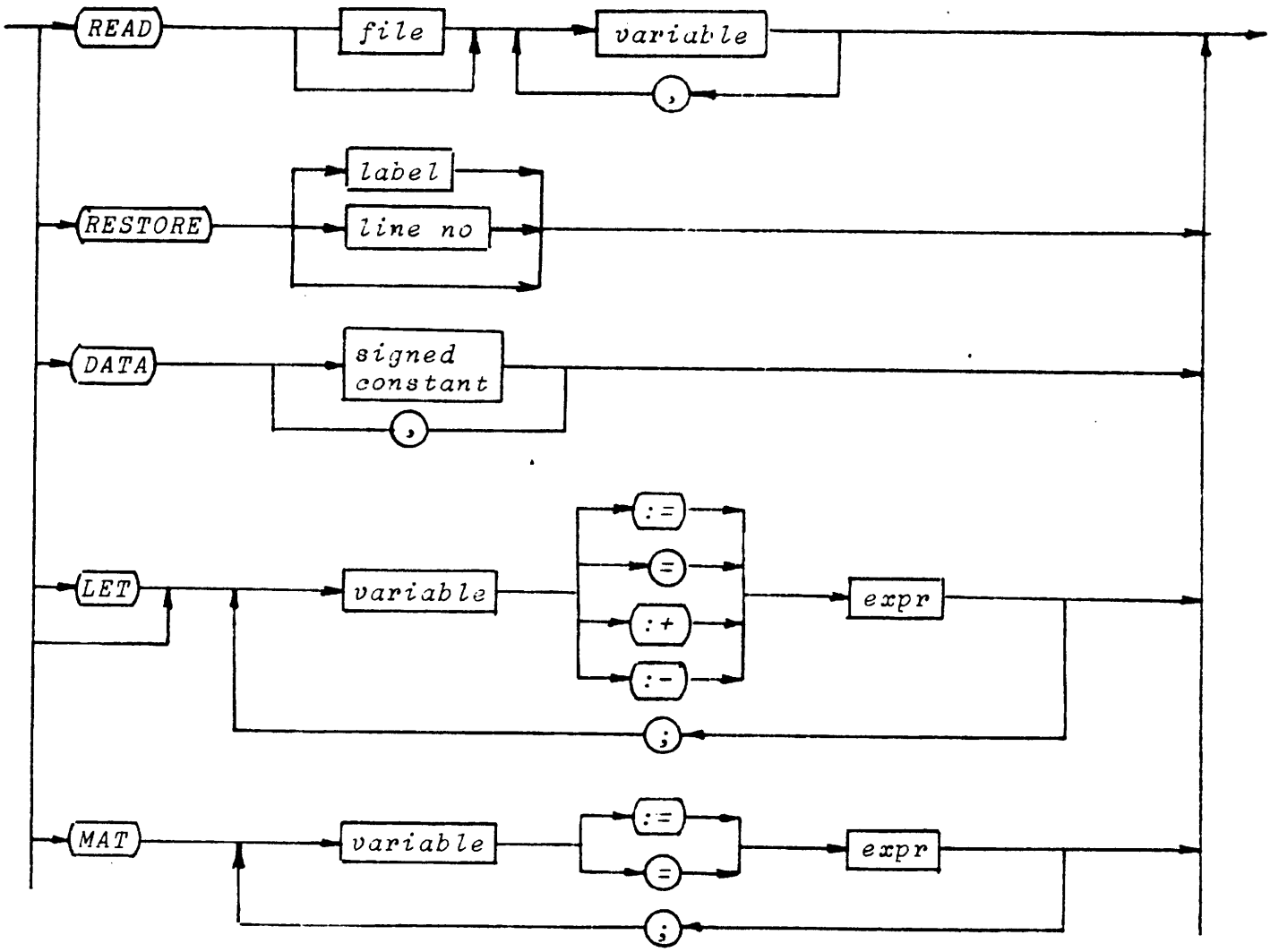
# COMAL 80

## SYNTAX DIAGRAMS.

*line:*




*statement:*




Note:

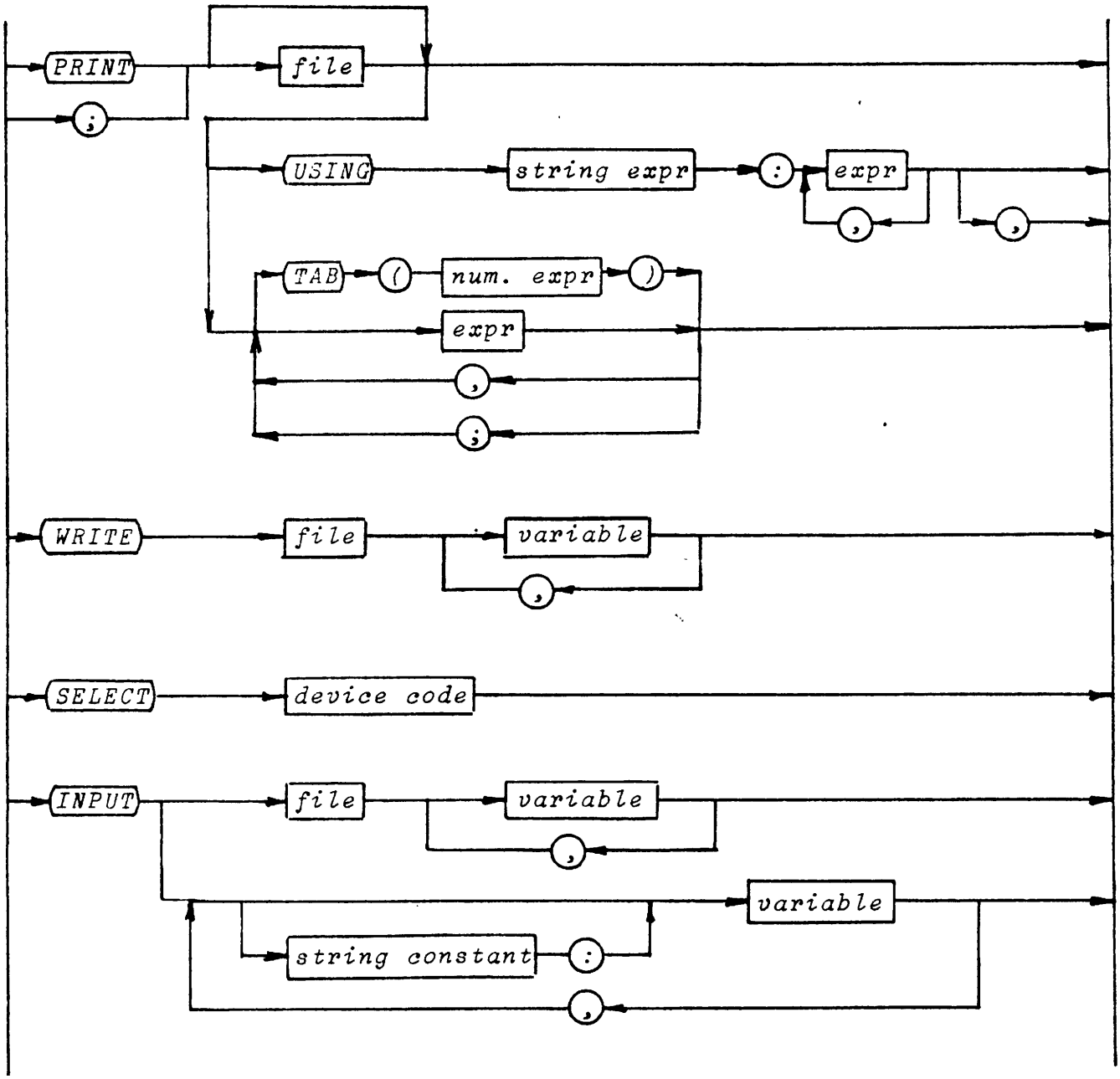
:+ may be used with strings, whereas :- may not.

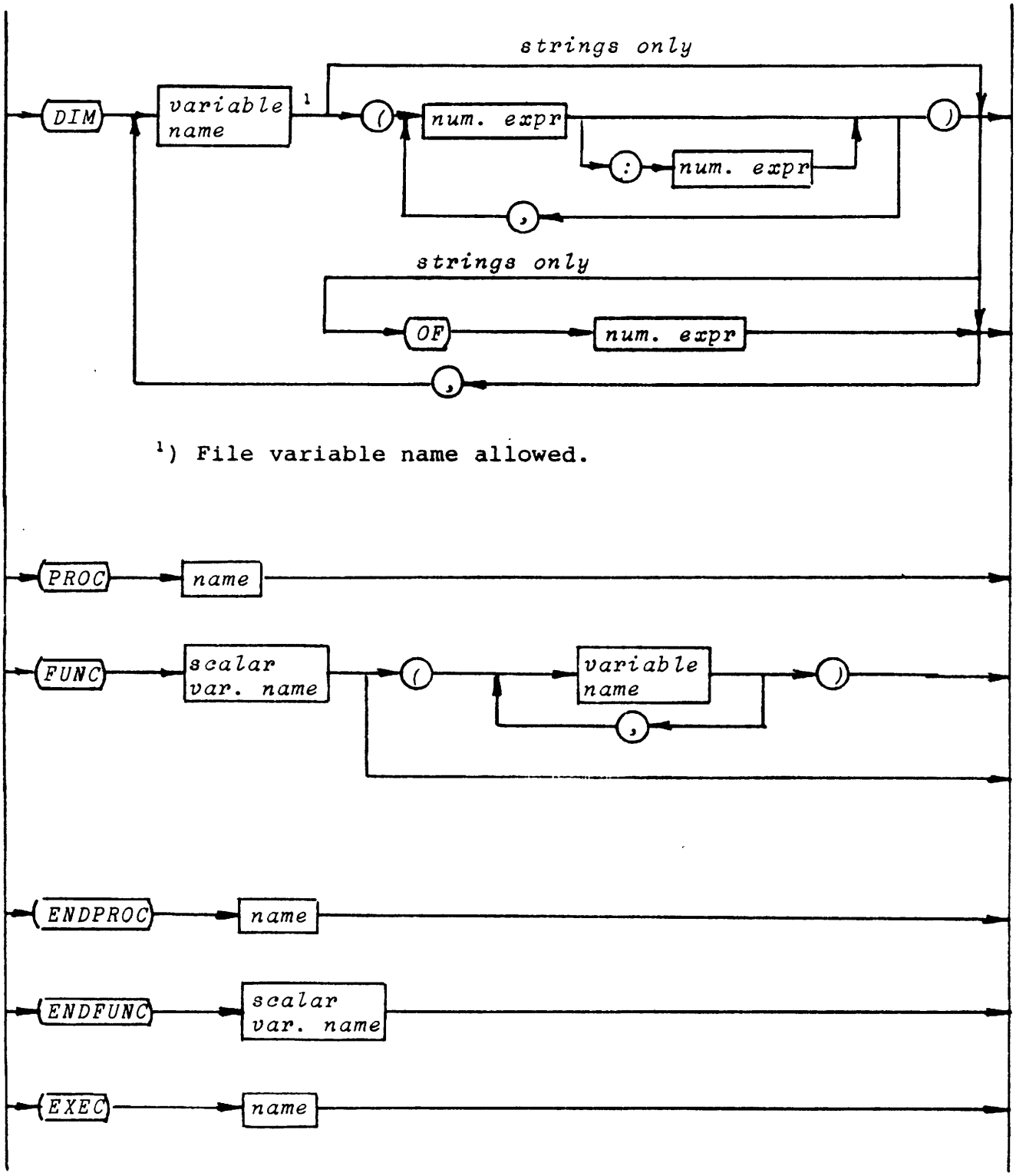
(note cont'd)

Variable and expression in an assignment must be of the same type. The only exception to that rule is:

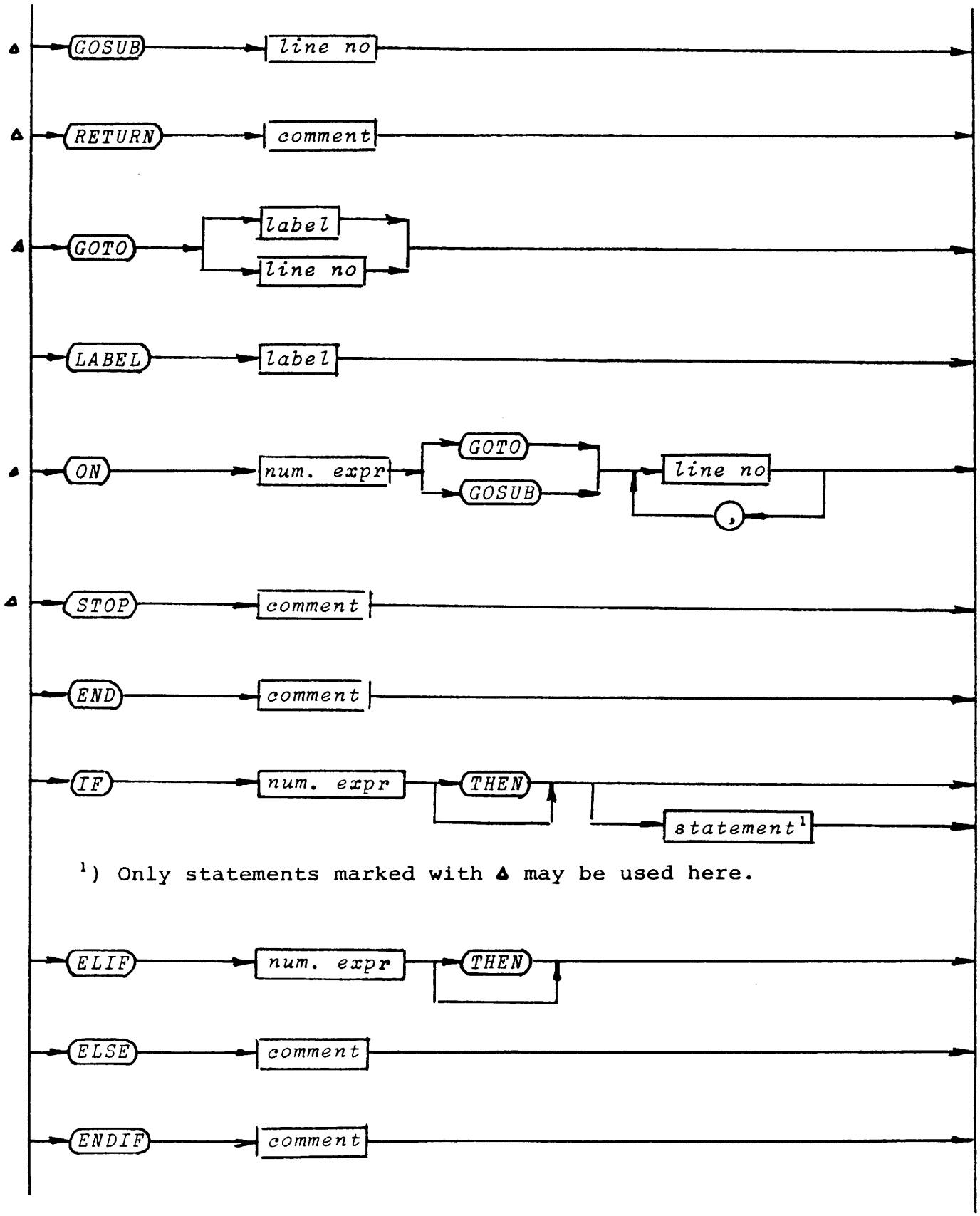
*real variable:=integer expression*

Variables must not be of type 'file' except where explicitely mentioned.

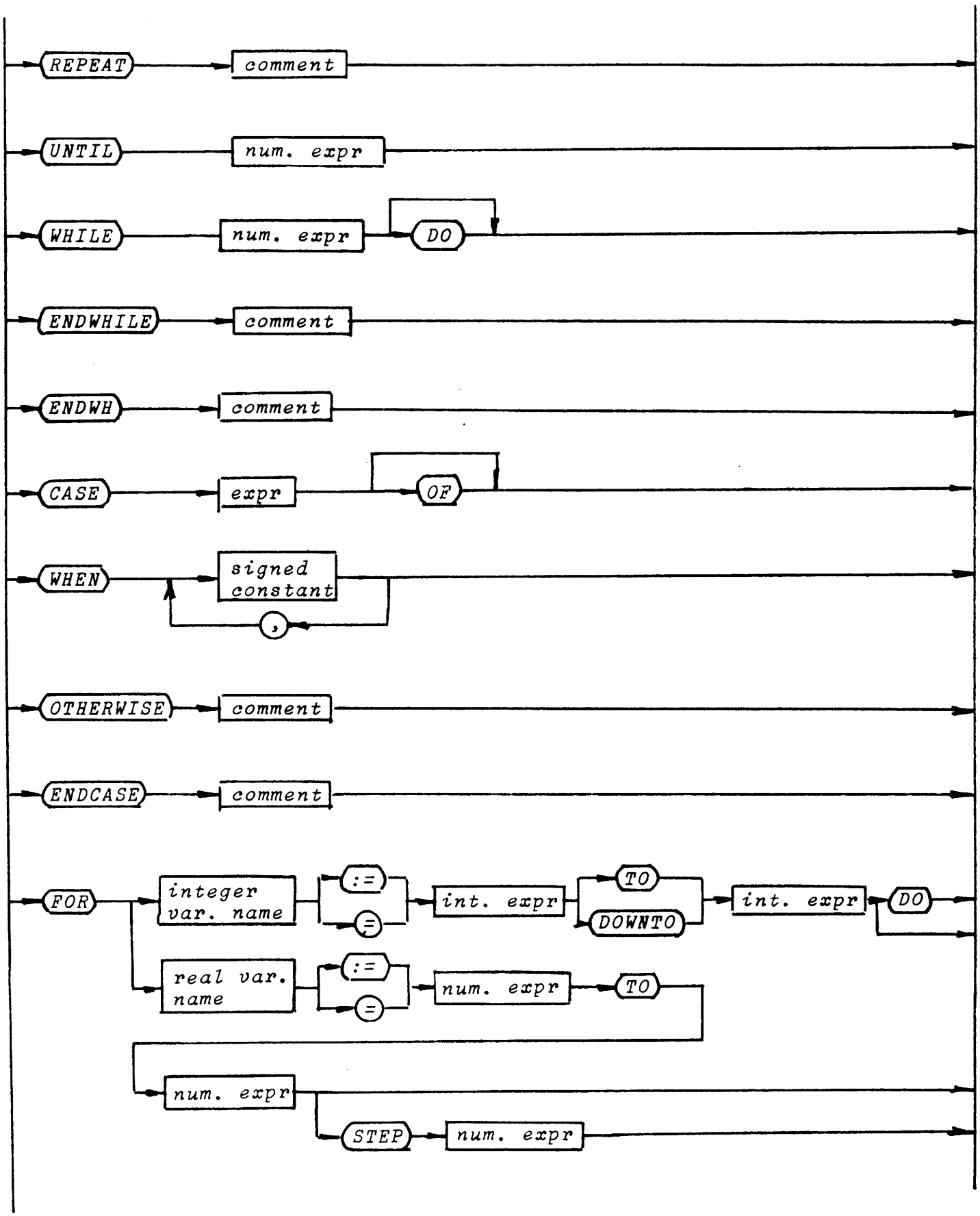


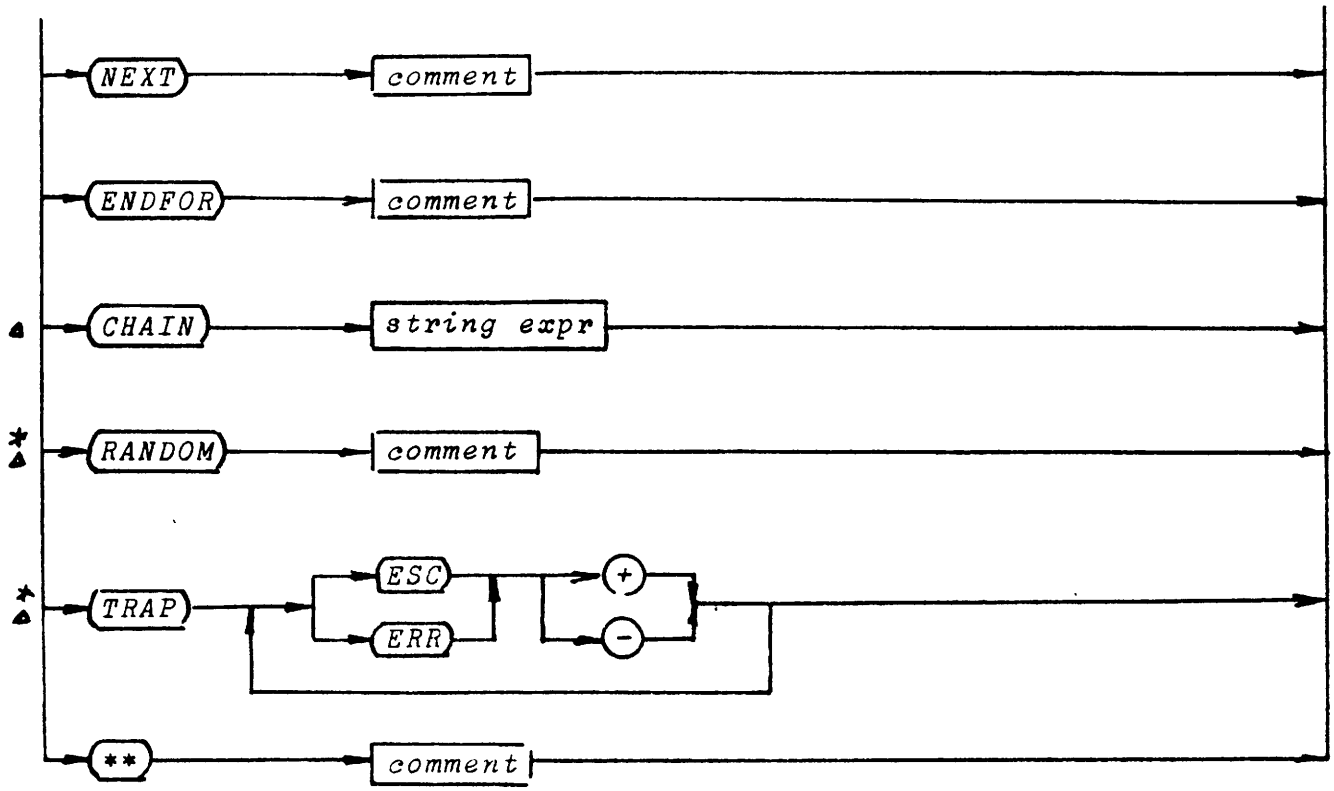



[1]) File variable name allowed.

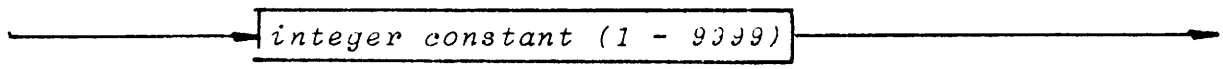



[1]) Only statements marked with ▲ may be used here.

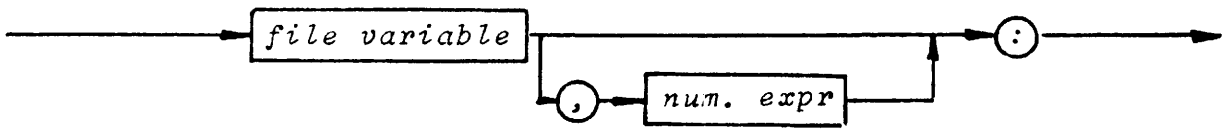

REPEAT
comment
UNTIL
num. expr
WHILE
num. expr
DO
ENDWHILE
comment
ENDWH
comment
CASE
expr
OF
WHEN
signed constant
,
OTHERWISE
comment
ENDCASE
comment
FOR
integer var. name
:=
=
int. expr
TO
DOWNTO
int. expr
DO
real var. name
:=
=
num. expr
TO
num. expr
STEP
num. expr




*line no:*



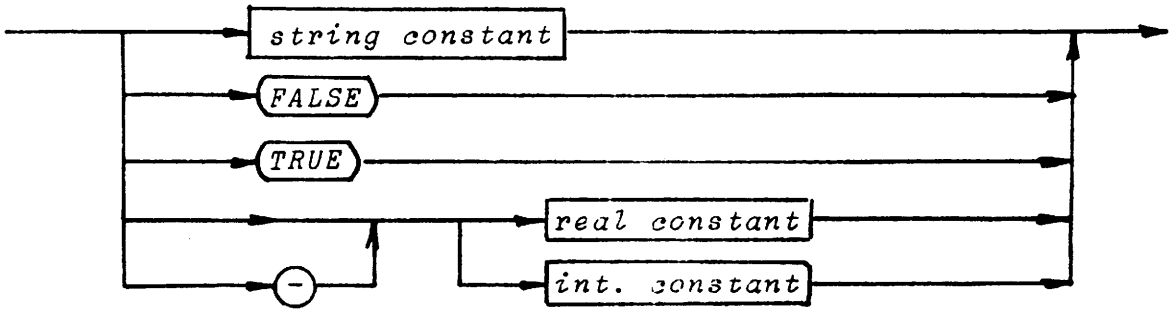

*file:*



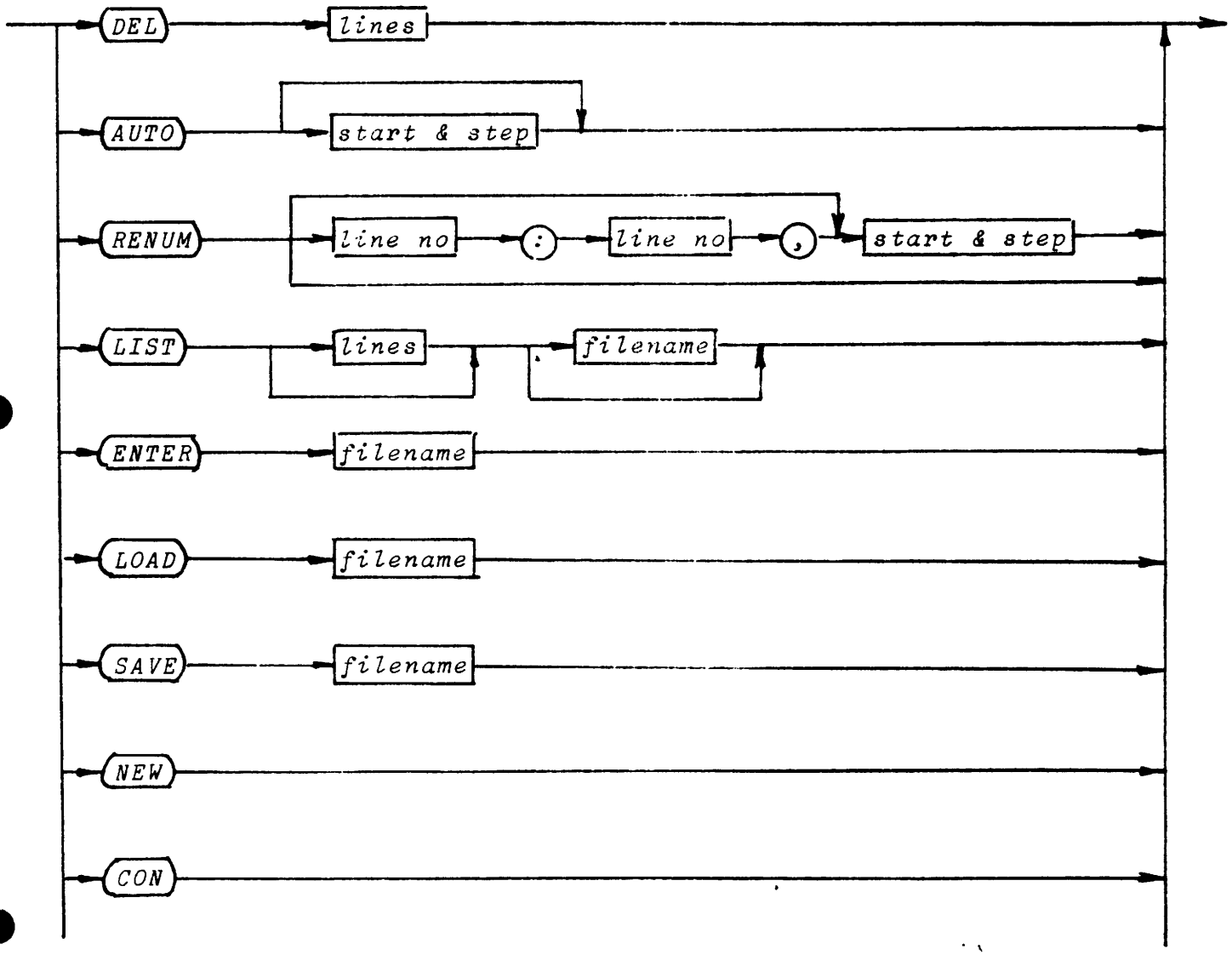

*label:*

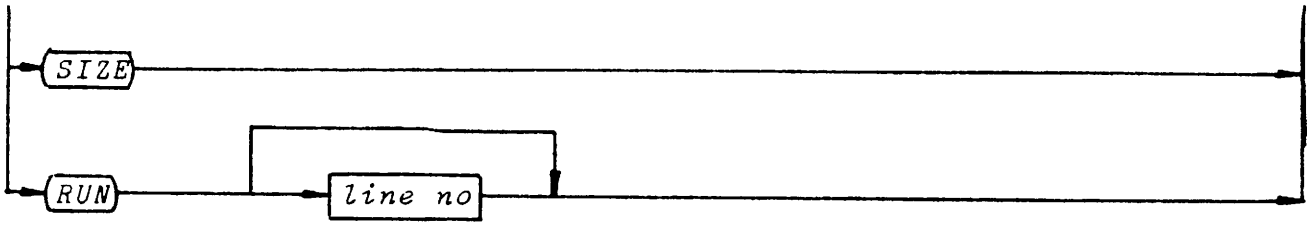


*signed constant:*




*command:*

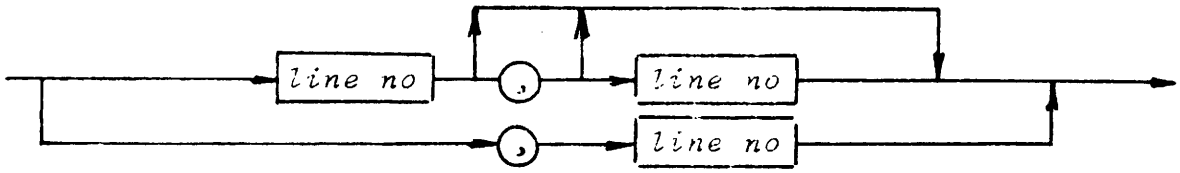




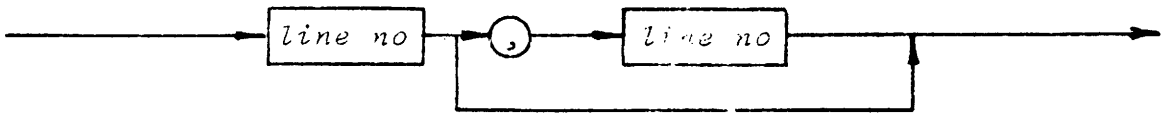

Note:

All statements marked with * may be used as commands.

*lines:*



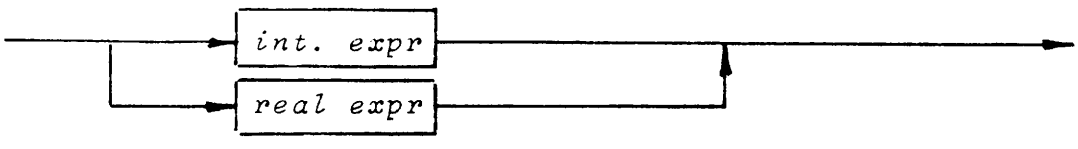

*start & step:*



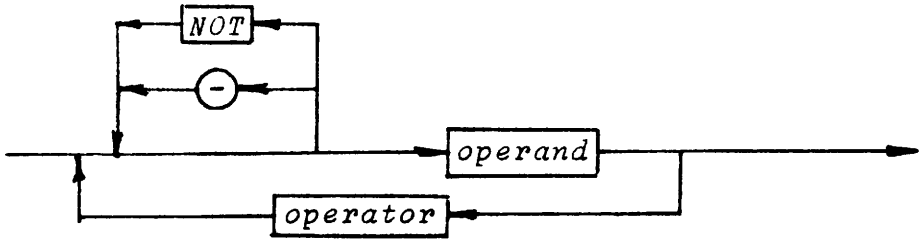

*filename:*

*any sequence of characters not starting with a digit, a comma, a blank, or a colon.*

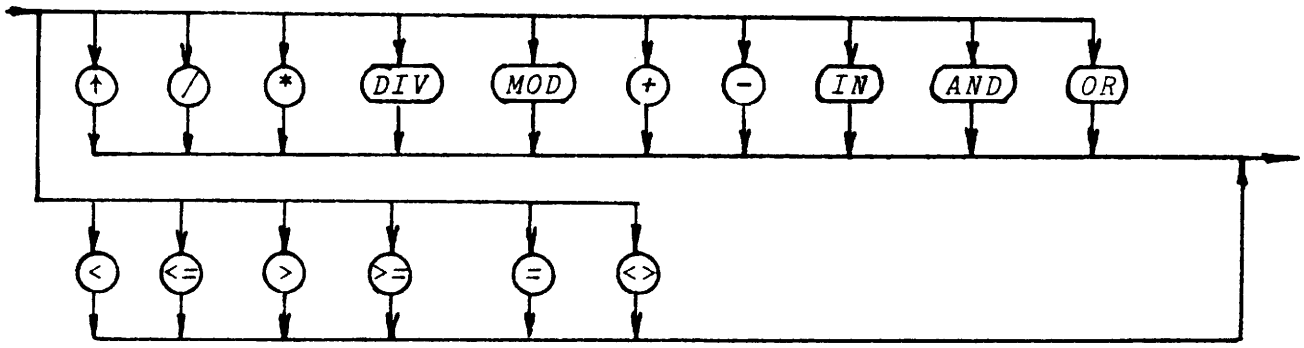
*num. expr:*



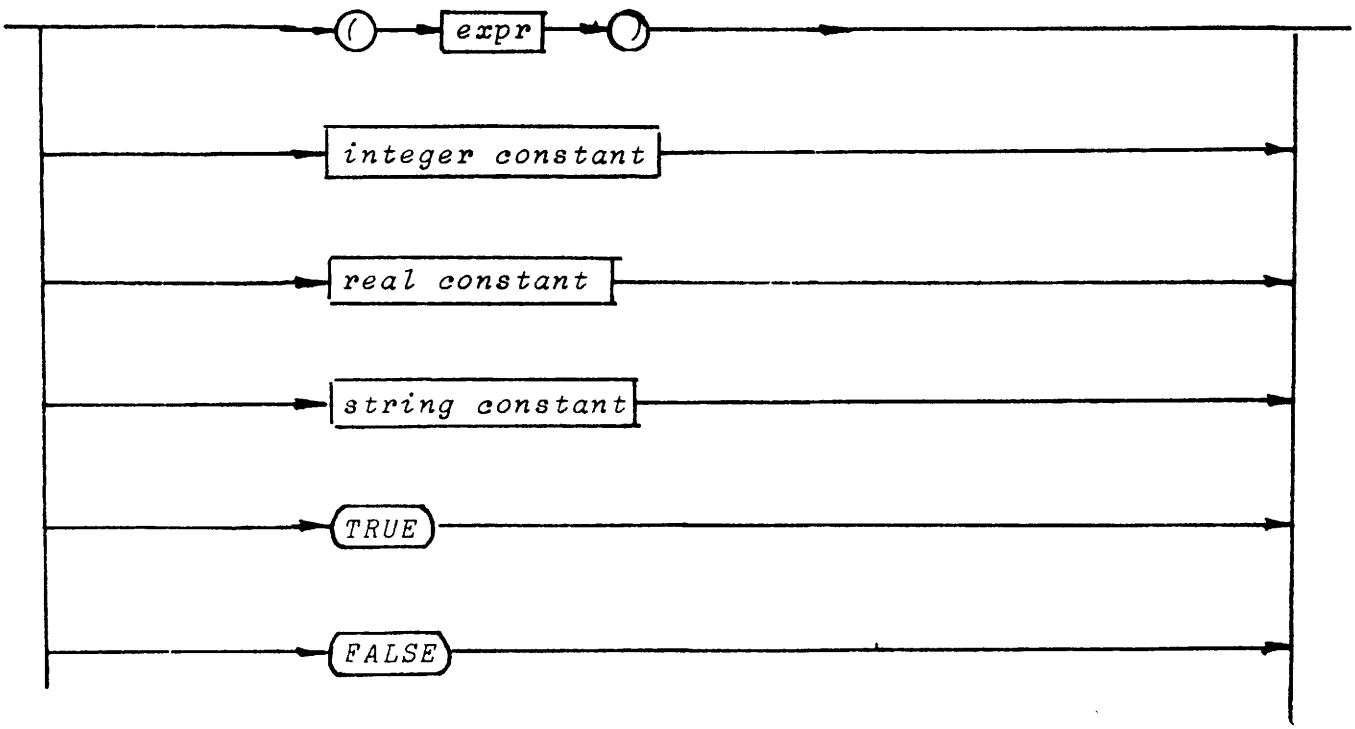
*(string, int, real, bool) expr:*

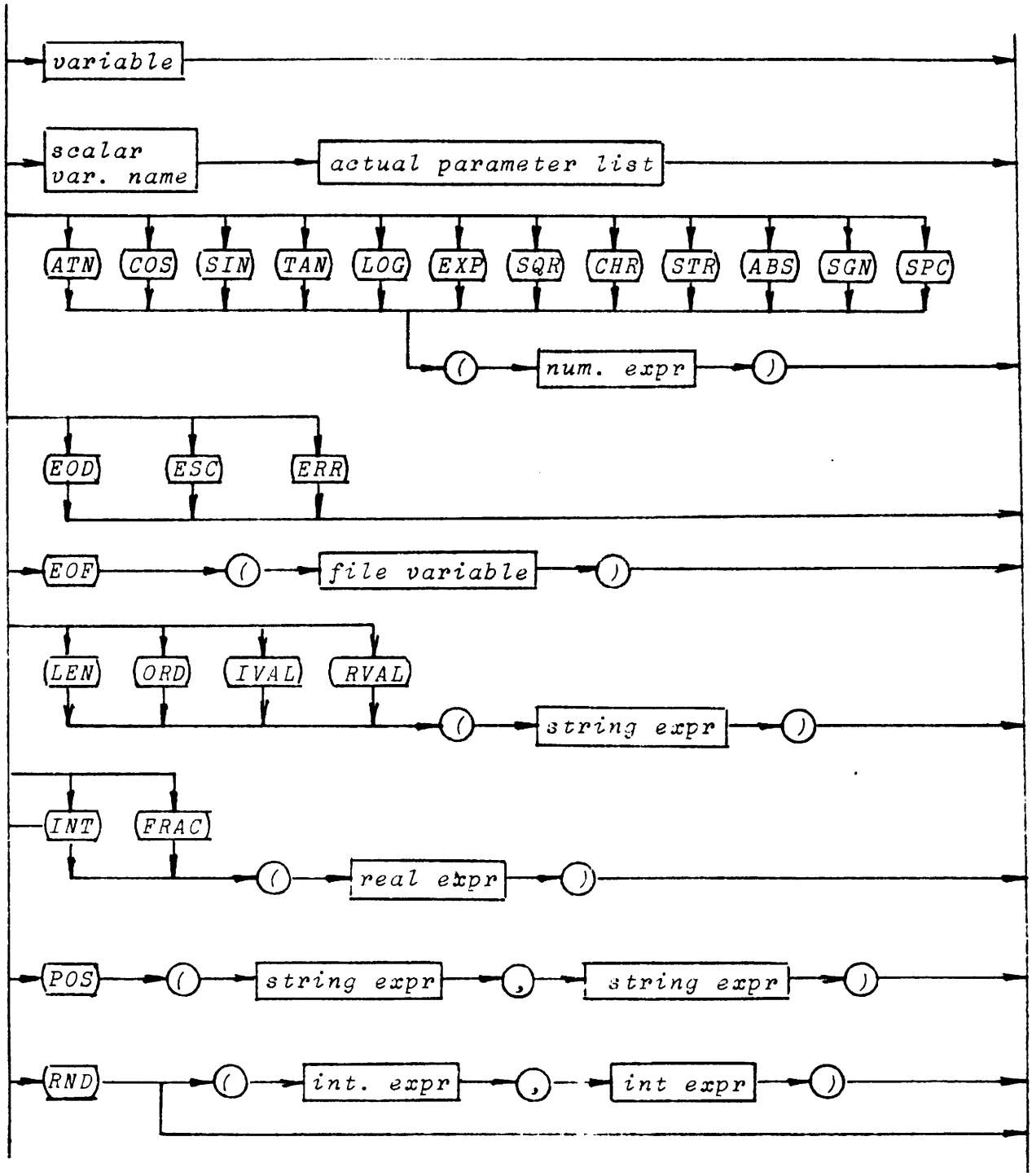



*operator:*

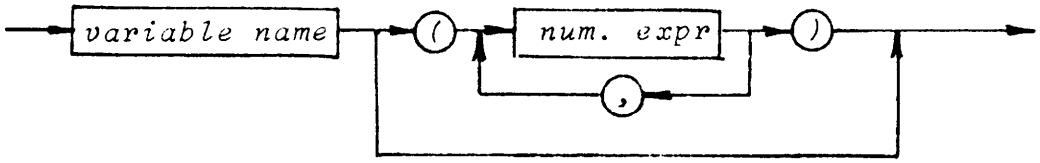



*operand:*

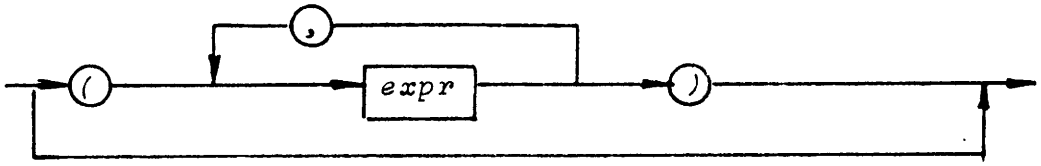



variable
scalar var. name
actual parameter list
ATN
COS
SIN
TAN
LOG
EXP
SQR
CHR
STR
ABS
SGN
SPC
(
num. expr
)
EOD
ESC
ERR
EOF
(
file variable
)
LEN
ORD
IVAL
RVAL
(
string expr
)
INT
FRAC
(
real expr
)
POS
(
string expr
,
string expr
)
RND
(
int. expr
,
int expr
)

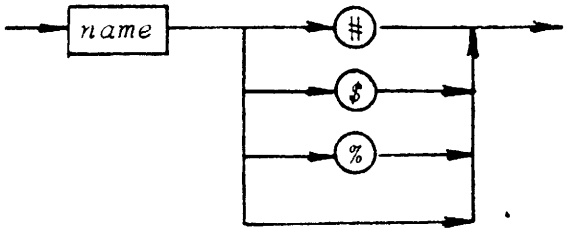
*variable:*




*actual parameter list:*




*variable name:*



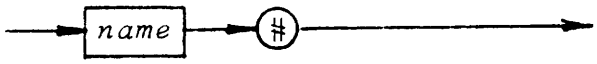

(integer)

(string)

(file)

(real)

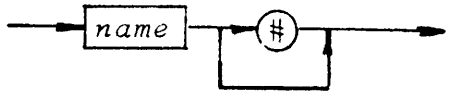
*integer variable name:*



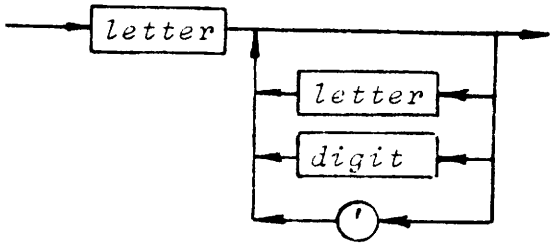

*real variable name:*



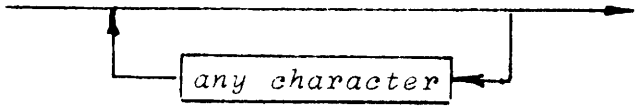
*scalar variable name:*



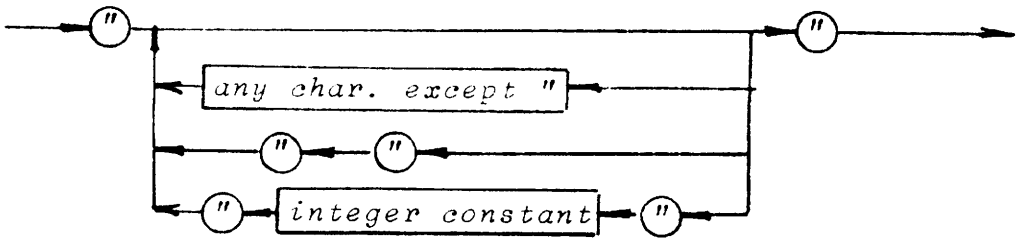

*name:*




*comment:*




*string constant:*



# COMMENTS TO

## THE SYNTAX DIAGRAMS

### INTRODUCTION

COMAL 80 includes COMAL 75 and a version of BASIC. If two COMAL 80 versions differ, it will always be on the BASIC part, since the COMAL extensions are well defined. Only structured BASIC which includes these extensions may be called COMAL. The COMAL 80 extensions of COMAL 75 have been carefully designed to meet the needs of users as observed through four years work with COMAL 75.

The following exposition will deal mainly with the statements that define COMAL. BASIC statements will be mentioned only where they have been subject to changes due to the definition of COMAL. Since the COMAL statements have been introduced to facilitate structured programming, the syntax diagrams can only unveil very little of the true power of these statements that has to be seen in global contexts. Great care will therefore be taken to display the structures controlled by the most important COMAL statements.

### LET

For educational purposes assignment in COMAL may be denoted by the symbol

```
:=
```

For reasons of compatibility it is, however, allowed to use an ordinary sign of equality when typing in the program. The interpreter will automatically convert this sign to :=.

The symbols

```
:+   and   :-
```

may be used in assignments where the same variable appears on both sides. Thus

```
NUMBER:+1
```

is equivalent to

```
NUMBER:=NUMBER+1
```

A LET statement will take as many assignments as the line width permits, each individual assignment being separated from the next one by means of the semicolon (;).

MAT

May be used to assign values to all components in an array. Thus the statement

```
MAT ACCOUNT:=0; FOUND#:=FALSE
```

where ACCOUNT is an array of reals and FOUND# an array of integers, will assign a value of 0 to each component of ACCOUNT and a value of FALSE (=0) to each component of FOUND#.

SELECT

May be used as command or statement. Device code could be for example: LPT (lineprinter), TTY (teletype), or PTP (punch). The statement

```
SELECT LPT
```

causes the output from all following PRINT statements to be sent to the lineprinter. The statement

```
SELECT TTY
```

resets the function back to normal teletype output. Output from string constants in INPUT statements is not affected by the SELECT.

EXEC, PROC, ENDPROC

If part of a program is initiated with the statement

PROC *name*

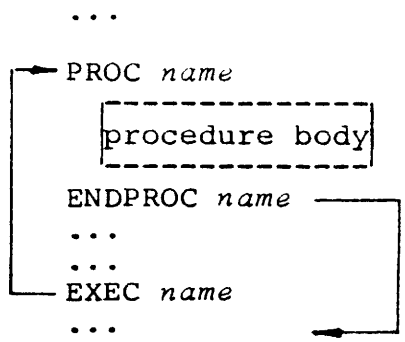
where *name* is a string formatted as a variable name, and is terminated with the statement

ENDPROC *name*

this program may be called as a subroutine by another program using the statement

EXEC *name*

When the subroutine has been executed, control is passed to the statement following the EXEC statement that called the subroutine.
The program text between the PROC and ENDPROC statements is indented in the program listing.




FUNC, ENDFUNC

If a subprogram is initiated with the FUNC statement and terminated with the ENDFUNC statement, it may be used by another program as a predefined function. All variables introduced in the lines between FUNC and ENDFUNC are local and global variables cannot be accessed from these lines. Parameters may be simple variables of any type, and they are all called by value.
The output from the function is returned through the functions name (i.e. the scalar variable name immediately following the FUNC keyword). Thus an assignment like this:

*function name:=expression of correct type*

must appear somewhere in the body of the function.

Example.

```
FUNC GCD#(X#,Y#)
  ...
  ...
  GCD#:=A#
ENDFUNC GCD#
```

This function is used in the statement:

```
IF GCD#(A#,B#)=1 THEN PRINT "A AND B ARE REL. PRIMES."
```

□□□

GOTO, LABEL

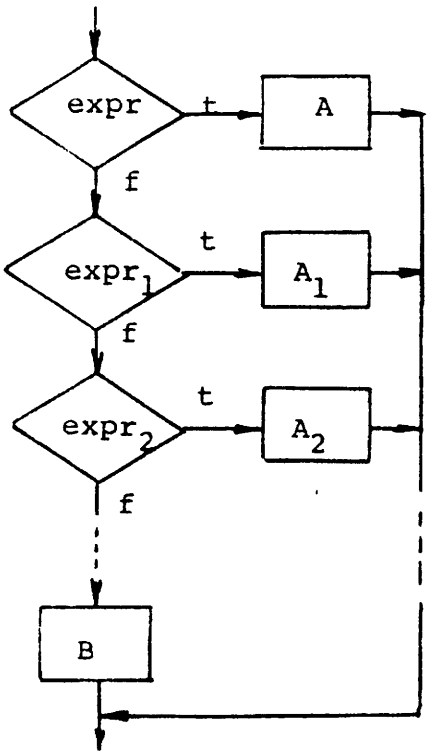
Adresses for GOTO may be given by labels in COMAL. Also the RESTORE statement may use a label. Thus the statement:

```
RESTORE NAMES'OF'PERSONS
```

will set the data pointer to the first element in the queue

trigger off a process.

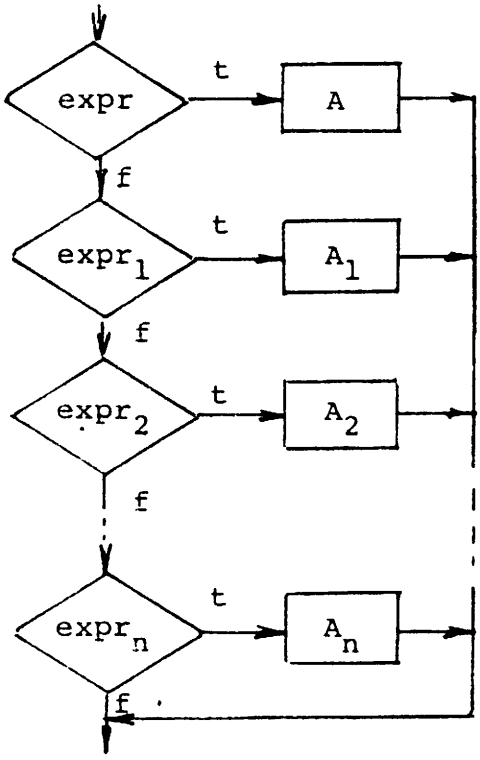
```
IF expr THEN
    [ A ]
ELIF expr1 THEN
    [ A1 ]
ELIF expr2 THEN
    [ A2 ]
...
...
ELSE
    [ B ]
ENDIF
```




If the final alternative ELSE is left out, you get

d. IF .. ELIF .. ELIF .. .. ENDIF

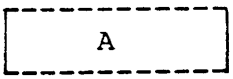
```
IF expr THEN
    [ A ]
ELIF expr1 THEN
    [ A1 ]
ELIF expr2 THEN
    [ A2 ]
...
...
ELIF exprn THEN
    [ An ]
ENDIF
```



REPEAT, UNTIL

The REPEAT and UNTIL statements provide the following structure:

REPEAT



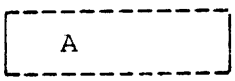

UNTIL expr

Program section A is executed repetitively until the expression following UNTIL has a value equivalent to true.
When this happens, control passes to the statement following the UNTIL statement.
The program text between REPEAT and UNTIL is indented in the program listing.

WHILE, ENDWHILE, ENDWH

The WHILE and ENDWHILE (ENDWH) statements provide the following structure:

WHILE expr DO




ENDWHILE (ENDWH)

Program section A is executed repetitively while the expression following the WHILE keyword is evaluated to true. When the expression evaluates to false, control passes to the statement following ENDWHILE (ENDWH).
The program text between WHILE and ENDWHILE (ENDWH) statements is indented in the program listing.

CASE, WHEN, OTHERWISE, ENDCASE

(diagram on next page)

When the expression following CASE has been evaluated, the list following the first WHEN is examined. If one of the constants in this list is equal to the value of the expression, program section $A_1$ is executed, and control is then passed to the statement following ENDCASE. If no such item is found, the list following the second WHEN is examined. If the value of the expression is found, $A_2$ is executed,

and control is then passed to the statement following ENDCASE. If the value still has not been found, the interpreter starts on the third list etc.
A default case (program section B) may be inserted and is executed if the value of the expression is not found in any of the lists following the WHEN keywords. The default case is indicated by the keyword OTHERWISE.

```
CASE expr OF
WHEN list1
   [    A1    ]
WHEN list2
   [    A2    ]
...
...
WHEN listn
   [    An    ]
OTHERWISE
   [    B     ]
ENDCASE
```

The OTHERWISE case may be left out, but the interpreter will then stop the execution of the program with an error message if no constant corresponding to the value of the expression has been found in the WHEN lists.
Note that at most one of the cases is executed. If it so happens that the value of the expression may be found in more than one of the lists, only the first of these lists will trigger off its process.
The program texts $A_1$, $A_2$, ..., $A_n$, B are indented in the program listing.

FOR, ENDFOR

The FOR .. NEXT loop structure from BASIC has been extended. As seen from the syntax diagram, you may use a statement like this:

```
FOR I#:=10 DOWNTO 1 DO
```

The "stepvalue" is then automatically set to -1. FOR loops with integers are very fast.
ENDFOR may used for NEXT. The countervariable may or may not occur after NEXT and ENDFOR. The interpreter will in any case look upon it as a comment.

## TRAP, ESC, ERR

Two dedicated flags ESC and ERR may be set or reset using the TRAP statement. A + will set the flag, and a - will reset it. When the interpreter starts, the two flags are set, and that means that the ESC key will cause a break whenever striked, and that errors will cause an error message and a program stop. If, however, one of the flags is reset, the interpreter will not react to the said conditions unless this has been defined explicitely in the program. This may be done by statements like:

```
IF ESC THEN EXEC TEST02
```

or

```
WHILE NOT ERR DO
```

## COMMENTS

Since comments are allowed after any statement, directely or by using **, the REM statement is left out. It may of course be introduced in the BASIC part for compatibility if wished. This has nothing to do with the definition of COMAL.

## TRUE, FALSE

To improve the readability of the programs two constants TRUE and FALSE are predefined. TRUE is equvalent to 1, and FALSE is equivalent to 0.

## AND, OR, NOT

In COMAL you have full Boolean algebra at your disposal. As mentioned before a numerical expression is in proper context considered false, if it has a value of 0, and true in all other cases. A Boolean expression like

```
NUMBER>MAX'NUMBER OR NOMORE
```

will output a value of 1, if it is true, and a value of 0, if it is false. A statement like this:

```
FOUND:=(NAME$=STUDENT'NAME$(I))
```

will assign a value of 1 to FOUND, if the condition to the right of the := is met, and a value of 0, if not. Thereafter FOUND may be used as if it were a Boolean variable. The "pseudo Boolean" values 0 and 1 are represented as integers (2 bytes) so it may speed up the program, if integer variables are used for "Boolean purposes".

IN

The expression:

```
NAME$ IN TEXT$
```

will output a value of 1, if NAME$ is found as a substring in TEXT$, and a value of 0, if it is not found.

Example

```
IF CH$ IN VOWELS$ THEN VOW#:=TRUE
```

□□□

NAMES

Variable names may contain as many characters as you wish. The first character must be a letter, the following may be letters, digits, or the sign '.

Example.

```
NUMBER'OF'STUDENS, MAXNUMBER, NUMBER, NAME$, NAME'OF'STD$
```

□□□

# APPENDIX 01

Survey of the data types which the different operators may work on, and the resulting type.

| left operand | right operand | operator | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | ↑ | / | * | DIV MOD | +- | IN | AND OR |
| str | str | | | | | str* | int | |
| int | int | real | real | int | int | int | | int |
| int | real | real | real | real | real | real | | int |
| real | int | real | real | real | real | real | | int |
| real | real | real | real | real | real | real | | int |

The relational operators: < <= > >= = <>
may work on any pair of strings and any pair of numerical expressions. The output will be an integer 1 or an integer 0. ("pseudo true" and "pseudo false").

The blank positions in the table mean that the corresponding operator may not be used with the set of operands.

*) Not -

Standard functions.

ATN, COS, SIN, TAN, LOG, EXP, SQR, FRAC, RVAL: real

CHR, STR: string

SGN, LEN, ORD, IVAL, INT, POS: int

EOD, ESC, ERR, EOF: int

ABS: same type as argument

RND: { without arguments: real
{ with arguments: int (arg. gives limits).

SPC: outputs a string which consists of as many blanks as the argument gives.

SAMPLE PROGRAM

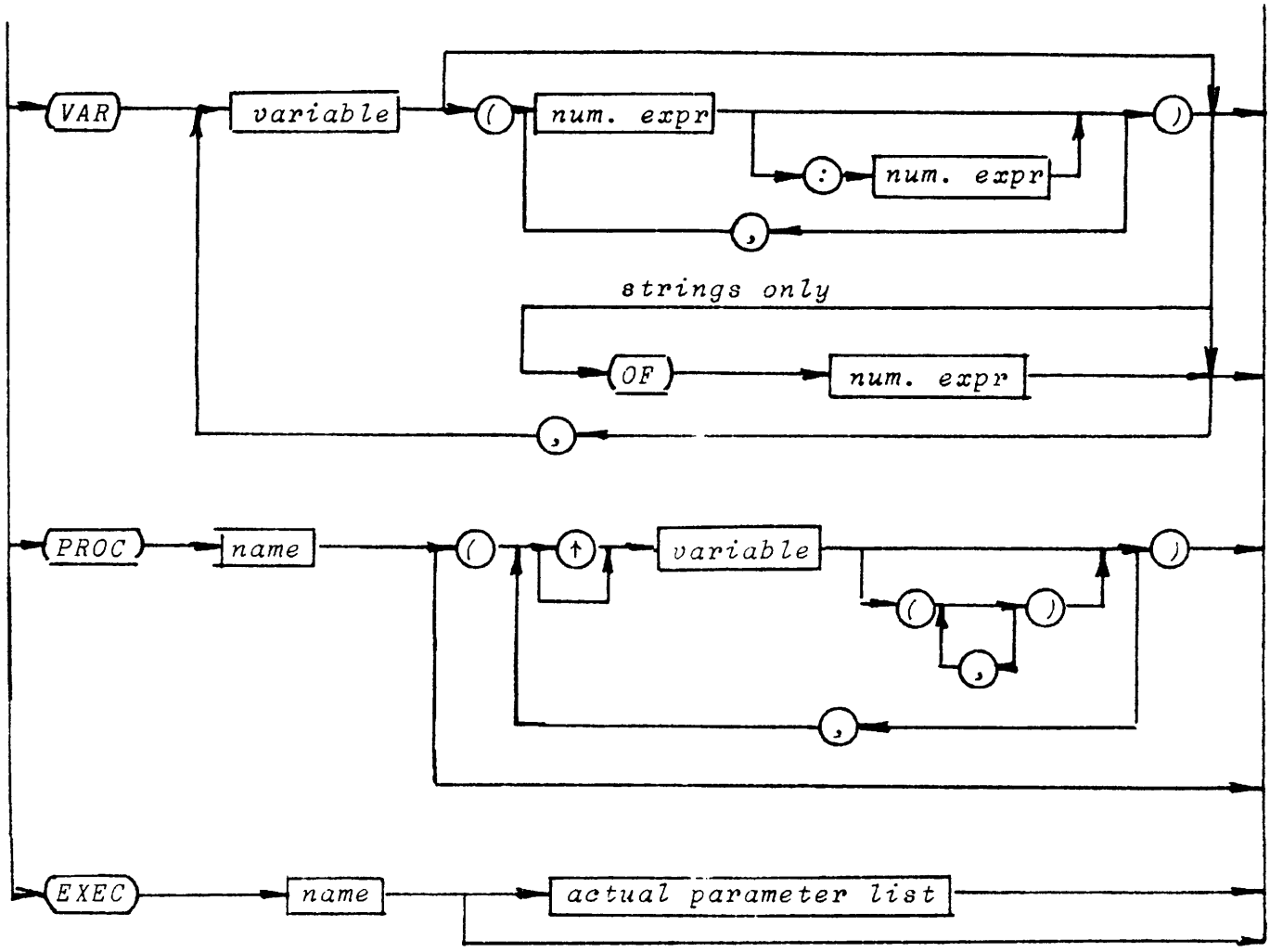
```
0010 ** THE SIMULATOR: MULTI-CASINO **
0020 ** WRITTEN IN COMAL 80 **
0030 ** BY BORGE R. CHRISTENSEN **
0040 ** AT 'DATO', TONDER, DENMARK **
0050 ** DATE OF THIS VERSION: JUNE 22, 1979 **
0060 ****
0070 **----------------**
0080 ****
0110 RANDOM
0120 FUNC BADBOY#(X#)
0130   BADBOY#:=(X#>=4)
0140 ENDFUNC BADBOY#
0150 **----------------**
0160 ** ATTRIBUTES OF CASINO ARE INITIALIZED **
0170 EMPTY:=TRUE; FULL:=FALSE
0180 **
0190 ** ATTRIBUTES OF GAMBLERS ARE INITIALIZED **
0200 DIM ACTIVE#(10), GOING#(10), REALBAD#(10)
0210 DIM WARNINGS#(10), BET(10), ACCOUNT(10)
0220 MAT ACTIVE#:=FALSE; GOING#:=FALSE; REALBAD#:=FALSE
0230 MAT WARNINGS#:=0; BET:=0; ACCOUNT:=0
0250 ****
0260 ** UTILITY STRINGS ARE DECLARED **
0270 DIM ANSW$ OF 5, COLOUR$(10) OF 6
0275 DIM OUTCOME$ OF 6, NAME$(10) OF 20
0280 ** ---------------- **
0290 ** ENDINIT **
0310 ****
0320 ** MAINPROGRAM **
0330 ****
0340 REPEAT
0350   IF NOT FULL THEN
0360     PRINT
0370     PRINT
0380     INPUT "NEW GAMBLERS? ": ANSW$
0390     IF ANSW$(1)="Y" THEN EXEC INREG
0400   ENDIF
0410   FOR I#:=1 TO 10 DO
0420     PRINT
0430     IF ACTIVE#(I#) THEN PRINT "YOUR TURN ";NAME$(I#)
0440     IF ACTIVE#(I#) THEN EXEC GUESS
0450     IF ACTIVE#(I#) THEN EXEC BET
0460     IF GOING#(I#) THEN EXEC BYEBYE
0470   ENDFOR
0480   IF NOT EMPTY THEN
0490     EXEC WHEEL
0500     FOR I#:=1 TO 10 DO
0510       IF ACTIVE#(I#) THEN EXEC STATUS
0520       IF GOING#(I#) THEN EXEC BYEBYE
0530     ENDFOR
0540   ENDIF
0550 UNTIL EMPTY
0560 PRINT "NO MORE GAMBLERS."
0570 PRINT "CASINO WILL BE OFF, UNTIL NEW GAMBLERS ARRIVE."
0580 PRINT "BYE - BYE!"
0590 END OF MAIN
```

```
0670 PROC GUESS
0680   OK:=FALSE
0690   REPEAT
0700     PRINT
0710     PRINT "WHAT COLOUR DO BET ON? "
0720     INPUT "BLU(E)/GRE(EN)/YEL(LOW)/BLA(CK)/RED  ": COLOUR$(I%)
0730     CASE COLOUR$(I,1,3) OF
0740     WHEN "NON"
0750       GOING%(I%):=TRUE; ACTIVE%(I%):=FALSE
0760     WHEN "BLU","GRE","YEL","BLA","RED"
0770       OK:=TRUE
0780     OTHERWISE
0790       PRINT
0800       PRINT "OPERATING ERROR! IMPOSSIBLE SITUATION!"
0810       INPUT "WANT INSTRUCTION? (YES/RETURN) ": ANSW$
0820       IF NOT ANSW$="" THEN EXEC INSTR
0830     ENDCASE
0840   UNTIL OK OR GOING%(I%)
0845 ENDPROC GUESS
0850 ** **
0860 ** ----------------**
0870 ** **
0880 ** BANKER'S TASKS **
0890 ** **
0910 PROC ACCOUNT
0920   REPEAT
0930     OK:=FALSE
0940     PRINT
0950     INPUT "HOW MUCH DO YOU WANT TO INVEST? ": INVEST
0960     IF INVEST<0 THEN
0970       PRINT
0980       PRINT "KEEP YOUR FALSE MONEY - YOU!"
0990       WARNINGS%(I%):+1
1000     ELIF INVEST=0 THEN
1010       PRINT
1020       PRINT "I HAD THE IMPRESSION, YOU MEANT BUSINESS!"
1030       WARNINGS%(I%):+1
1040     ELIF INVEST<1 THEN
1050       PRINT
1060       PRINT "NO SIR!! NOT THAT CENT STUFF. REAL MONEY PLEASE!"
1070       WARNINGS%(I%):+1
1080     ELIF INVEST<>INT(INVEST) THEN
1090       PRINT
1100       PRINT "TIPS! YOU A R E GENEROUS SIR! "
1110       INVEST:=INT(INVEST)
1120       OK:=TRUE
1130     ELSE
1140       OK:=TRUE
1150     ENDIF
1160     IF OK THEN ACCOUNT(I%):+INVEST
1170     GOING%(I%):=BADBOY%(WARNINGS%(I%))
1180   UNTIL OK OR GOING%(I%)
1190 ENDPROC ACCOUNT
```

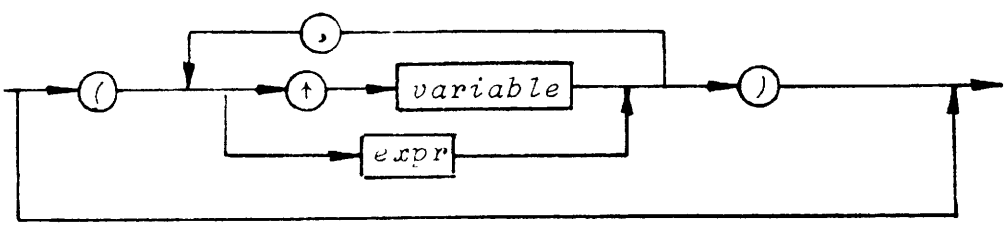
APPENDIX02

# XCOMAL 80

## SYNTAX DIAGRAMS




*actual parameter list:*



## VAR

The VAR statement may be used in stead of the DIM statement. Variables, declared by means of the VAR statement within the body of a procedure, are local to that procedure. Therefore simple variables may occur in the list following the keyword VAR.

Example

If the statement

```
VAR I,J,NUMBER,NAME$(40) OF 30
```

appears in the body of a procedure, the variables I and J, and the array NAME$ will be local to that procedure.

□□□

## PARAMETERS

Parameters for procedures may be called by value or by reference. If a parameter name begins with the character ↑, it is called by reference and otherwise it is called by value. If an array is refered to by a parameter, its dimension must be indicated by means of parentheses and commas. Thus

```
↑NUM(,)
```

indicates a parameter that refers to a two dimensional array of numbers and is called by reference.

Example

```
PROC SORT(↑ACCOUNT(),MAX)
```

The first parameter must be called by reference and the second one must be called by value. The first one will refer to a one dimensional array of numbers, the second one will be assigned the value of a number.

□□□

Procedures may be called recursively.