COMAL 80 PROGRAMMING LANGUAGE.

proposal

Background.

The first COMAL language was designed in 1974 by Børge R. Christensen, State Teachers' College, Tønder, Denmark, and Benedict Løfstedt, Department of Computer Science, University of Aarhus.

COMAL (COMmon Algorithmic Language) was constructed as an extension to BASIC, reflecting developments in programming languages and techniques, such as structured programming.

COMAL was intended primarily for use in public schools, but the language has found broader applications.

Since 1974 the language has been extended with additional facilities. Today several different versions of the language exist, supplied by a number of manufacturers.

In October 1979 a group interested in COMAL, including manufacturers, school teachers, and university people, concluded that a standardization of COMAL, in the form of COMAL 80, was needed. A working group was formed to write a proposal for the COMAL 80 language.

The standardization was to contain a nucleus forming the COMAL 80 language and recommendations for extending the language.

This report, written by the working group, is the proposal for the nucleus of the COMAL 80 language. Recommendations for extending the language will appear later.

The members of the working group included:

Arne Christensen, International Computers Limited A/S
Børge R. Christensen, Tønder State Teachers' College
Steffen Dyhrberg, International Computers Limited A/S
H. B. Hansen, Roskilde University Center
Rolf Molich, Dansk Data Elektronik Aps.
Jørgen Olsen, RC Computer A/S
Jesper Barfod, Digitek Data og Instrumentering
Tom Østerby (editor), Technical University of Denmark.

Other manufacturers, companies, and groups have followed the work with great interest.

General information.

COMAL 80 is a general purpose programming language intended for use by non-expert programmers.

COMAL 80 has facilities for writing structured programs.

The COMAL 80 language system is designed to operate in an interactive environment. COMAL statements are entered directly from the user's terminal and checked immediately for syntactical errors.

Execution of programs is also done in interactive mode. Normally the system will include facilities for debugging. These facilities are not part of the standard.

Description of the COMAL 80 language.

The language is described in the following order - program, statements, expressions, variables, constants, and characters. The description of each construction includes a general introduction, the syntactical format, the effect, and notes.

The syntax is specified by means of Backus-Naur notation, extended with the following notation:

{ } : meaning zero, one, or more occurrences
[ ] : meaning zero or one occurrences.

Notes contain remarks concerning use of the construction, including rules, precautions, operation, etc.

Space characters may be included where it is not specifically forbidden.

In the description of the language there are some places where the effect or result is undefined. An implementation might choose to give an error reaction here.

COMAL programs.

Format

```
<comal program> ::= <statement list>
<statement list> ::= { <statement> }
<statement> ::= <unstructured statement> ¦ <structured statement>
<unstructured statement> ::= <simple statement> ¦
                             <simple declaration statement>
<structured statement> ::= <compound statement> ¦
                           <structured declaration statement>
<simple statement> ::= <assignment statement> ¦ <io statement> ¦
                       <goto statement> ¦
                       <procedure call statement> ¦
                       <end statement> ¦ <stop statement>
<io statement> ::= <read statement> ¦ <restore statement> ¦
                   <input statement> ¦
                   <print statement> ¦ <print-using statement> ¦
                   <select output statement>
<compound statement> ::= <conditional statement> ¦
                         <repetitive statement>
<conditional statement> ::= <if statement> ¦ <case statement>
<repetitive statement> ::= <for statement> ¦ <while statement> ¦
                           <repeat statement>
<simple declaration statement> ::= <data declaration> ¦
                                   <label statement>
<data declaration> ::= <dim statement> ¦ <data statement>
<structured declaration statement> ::= <procedure declaration>
```

Input format.

A COMAL program consists of a number of statements. A statement is written on one or more program lines depending on the type of statement.

A program line may begin with a line number and may end with a comment.

If line number is present, then it must be an integer in the range 1 to 9999. The line numbers determine the order in which statements are stored in the program. The line number must be followed by at least one space character. A program line may consist of a line number only. Execution of such a program line has no effect.

A program line may end with a comment. The comment begins with two consecutive slashes (//), and may be followed by any sequence of displayable ASCII characters. A comment is stored, but has no effect on statement execution. Consecutive program lines in a structured statement must have increasing line numbers.

A <simple statement> must be contained within one program line. A program line must contain only one <simple statement>.

A <structured statement> may extend over one or more program lines. In the Backus-Naur description of a structured statement each line must correspond to a separate program line.

When a program is run the statements in the program are normally executed in the order in which they are stored. Certain statements can alter the sequence of execution

Assigment statement.

A statement to assign values to numeric and string variables.

Format

```
<assignment statement> ::=  <assignment list>
<assignment list> ::= <assignment> {;<assignment>}
<assignment> ::= <numeric assignment> | <string assignment>
<numeric assignment> ::=
              <left side> := <arithmetic expression>
<left side> ::= <numeric variable> | <procedure identifier>
<string assignment> ::=
        <string variable> := <string expression>
```

Statement execution.
---

a. The reference to the variable on the left side of ":=" is computed.
b. The expression, string or numeric, is evaluated.
c. The result of step b is assigned to the variable on the left side of ':='.
d. Steps a, b, and c are repeated for all assignments in the assignment list.

Notes.
---

1. In an assignment the type of variable and expression must be the same, either numeric or string.
2. If <string expression> is longer than the string variable, the string expression is right truncated to the length of the variable. If the string expression is shorter than the string variable, the string expression is placed left justified in the string variable or the substring. If the string variable is not a substring, the rest of the string variable is filled with characters indicating 'null' values. If the string variable is a substring, and the string expression is shorter than the substring, the rest of the substring is filled with blanks.
3. If the string variable is a substring, the start position of the substring must be less than or equal to the length of the string plus one (start postion <= LEN(string variable) + 1), otherwise the result is undefined.

READ statement.

A statement, applied to read values from the list defined by one or more DATA statements, to assign values to string and numeric variables listed in the READ statement.

Format

<read statement> ::= READ <variable> {,<variable>}
<variable> ::= <numeric variable> | <string variable>

Notes.
---

1. READ statements are always used in conjunction with DATA statements.
2. The variables listed in the READ statement may be subscripted or simple numeric or string variables.
3. A data element pointer is moved to the next available value in the DATA statement list as values are retrieved for variables in the READ statement. If the number of variables in the READ statement exceeds the number of values in the DATA statement list, a run-time error occurs.
4. The type (numeric or string) of a READ statement variable must match the type of the corresponding data element value; otherwise a run-time error occurs.
5. Reading a value to a string variable follows the same rules as described in the assignment statement.

RESTORE statement.

A statement to reset the data element pointer to the beginning of the DATA statement list.

Format

<restore> ::= RESTORE

INPUT statement.

A statement to assign values entered from the user's terminal during program execution to a list of string and/or numeric variables.

Format

<input statement> ::=
 INPUT [<string constant>:] <variable> {,<variable>} [<cont>]
<variable> ::= <numeric variable> | <string variable>
<cont> ::= ;

Statement execution.
---

a. A prompt character is output unless a string constant is included, in which case the value of the string constant is output. The standard prompt is a colon (:).
b. The user responds by typing a list of data items, which are assigned to the variables listed in the INPUT statement. The user ends the list of data items by activating the 'end-of-input' key.
c. If <cont> is specified, the printing position is left unchanged; otherwise the printing position will be the first position on the next line.

Notes.
---

1. Data entered must be of the same type (numeric or string) as the variable in the argument list for which the data is being supplied. Variables in the argument list may be subscripted or unsubscripted.
2. A data item supplied for a numeric variable can be typed in the following form:

   [+|-] <real number>

   One or more spaces can be typed before a numeric value.
3. A separator between two numeric values may be any character not part of a real number. The separator between a numeric value and a string is the first character not included in a real number.
4. If the data entered does not match the type of the current variable, an error will take place; the user can then enter data of the correct type.
5. If the 'end-of-input' key is activated before values have been assigned to all of the variables in the argument list, a prompt character will be output, indicating that further items are expected.
6. Entering a value for a string variable follows the same rules as described in the assignment statement.
7. 'End-of-input' is the only possible separator between two string values.

PRINT statement.

A statement to output values on the user's terminal.

Format

```
<print statement> ::= PRINT [ <print list> [ <print end> ]]
<print list> ::= <print element>
                     { <print separator> <print element> }
<print element> ::= <aritmetic expression> |
                    <string expression> | <tab function>
<print end> ::= <print separator>
<print separator> ::= , | ;
<tab function> ::= TAB ( <arithmetic expression> )
```

Notes.

1. The print line on a terminal is divided into print zones. The width of a print zone is not part of the standard.
2. If <print separator> is a comma (,) the output of the next element starts from the leftmost position of the next zone. If there are no more zones on the current line, printing continues in the first zone on the next line. If a print element requires more than one zone, the next element is printed in the next free zone. A print element is always printed on one line.
3. If <print separator> is a semicolon (;), the output of the next print element starts from the next character position. A space is printed after a number.
4. The effect of a <print end> is the same as a <print separator> (note 2 and 3). If no <print end> is specified, printing is continued on the first position on the next line.
5. A PRINT statement with no <print list> causes output of an empty line.
6. TAB(exp) is a function to tabulate the printing position for an item in the print list to the column number evaluated from 'exp'. Columns on the print line are numbered 1,2,.... If <arithmetic expression> evaluates to a column number greater than or equal to the current column number and less than or equal the length of the print line, the value of the expression indicates the new column position. If the equations are not satisfied, the effect of the function is undefined.

PRINT-USING statement.

A statement to output values of items using a specified format.

Format

```
<print-using statement> ::=
   PRINT USING <string expression> : <using list> [ <using end> ]
<using list> ::= <using element> { , <using element> }
<using element> ::= <arithmetic expression>
<using end> ::=  ;
```

Notes.

1. <string expression> is used as a format string. The characters in this string are treated in the following way:
   - `#` digit position and sign (sign position required only for negative numbers)
   - `.` decimal point (only if surrounded by #)

   All other characters in the format string are output directly.
2. The effect of <using end> is the same as descibed in note 4 for the PRINT statement.
3. String variables must not appear in the using list. However the value of a string variable can be output by concatenating it with the string expression.

SELECT OUTPUT statement.

A statement to control the device to which results from a program (PRINT or PRINT USING statement) shall be directed.

Format.

<select output statement> ::= SELECT OUTPUT <unit>
<unit> ::= <string expression>

Notes.

1. <string expression> specifies the output device. The names of output devices are not part of the standard.
2. All output from PRINT and PRINT USING will be directed to the unit specified until a new SELECT OUTPUT statement is executed.

GOTO statement.

A statement to transfer control unconditionally to another part of the program.

Format

<goto statement> ::= GOTO <label name>
<label name> ::= <identifier>

Notes.

1. A GOTO statement must not transfer control to a statement which is part of another structured statement or a procedure declaration.
2. A GOTO statement transferring control out of one or more structured statements will terminate these statements.
3. Execution of a GOTO statement out of a procedure will cause an error.

EXEC statement.

A statement to activate a procedure defined in a procedure declaration.

Format

```
<procedure call> ::=
     EXEC <procedure identifier> [ (<actual parameter list>) ]
<actual parameter list>::=
                     <actual parameter> { ,<actual parameter> }
<actual parameter> ::= <simple variable> ¦
                       <simple string name> ¦
                       <numeric array name> ¦
                       <string vector name> ¦
                       <arithmetic expression> ¦
                       <string expression>
```

Statement execution.

a. The procedure designated by <procedure identifier> is activated. Execution is started with the statement after PROC.
b. Execution is continued until an ENDPROC statement is encountered. Then execution is continued with the statement immediately following the EXEC statement.

Notes.

1. The number of actual parameters must be the same as the number of formal parameters in the procedure declaration.
2. The rules for substitution of the formal parameters by actual parameters are the following:

Formal parameter spec.	Actual parameter allowed
<simple variable>	<arithmetic expression>
REF <simple variable>	<simple variable>
<simple string name>	<string expression>
REF <simple string name>	<simple string name>
REF <numeric array name>	<numeric array name>
REF <string vector name>	<string vector name>

3. An actual parameter which is a <numeric array name> must have the same number of indices as specified for the formal parameter.

END statement.

A statement to terminate execution of the program and to return control to interactive mode.

Format

<end statement> ::= END

Note.

1. An END statement will terminate execution of the program. In an interactive environment a prompt is output on the user's terminal.

STOP statement.

A statement to stop execution of the program and to return control to the terminal in interactive mode.

Format

<stop statement> ::= STOP

Notes.

1. A STOP statement will terminate execution of the program. A stop indication, including the line number of the stop statement, will be output on the user's terminal.
2. In an interactive environment program execution may be resumed after a STOP statement.
3. In a batch environment execution of a STOP statement has the same effect as the END statement.

IF statement.

A statement to execute one of two blocks of statements depending on whether the value of a logical expression is true or false.

Format

```
<if statement> ::=
                IF <logical expression> THEN <simple statement> |
                IF <logical expression> THEN
                <statement list>
                [<else part>]
                ENDIF
<else part> ::= ELSE
                 <statement list>
```

Statement execution.

a. <logical expression> is evaluated.

b1. If the value is true, the <simple statement> after THEN is executed, otherwise the statement is skipped.

b2. If the value is true, the <statement list> between THEN and ELSE/ENDIF is executed.
If the value is false, the <statement list> after ELSE will be executed. If ELSE is not specified, execution will continue at the first statement following ENDIF.
If none of the executed statements cause transfer of control to another part of the program, execution will continue at the first statement following ENDIF.

CASE statement.

A statement to execute one of several blocks of statements depending on the value of an expression.

Format

```
<case statement> ::= CASE <expression> OF
                     <case list element>
                     {<case list element>}
                     [<otherwise part>]
                     ENDCASE
<case list element> ::= WHEN <case expression list>
                     <statement list>
<case expression list> ::= <expression> {,<expression>}
<otherwise part> ::= OTHERWISE
                     <statement list>
```

Statement execution

a. The expression after CASE is evaluated.
b. The expressions after WHEN are evaluated one by one until a value is found which is equal to the value obtained in step a.
c. If a matching value is found, the following statement list is executed until the next WHEN, OTHERWISE, or ENDCASE. After this, control is transferred to the first statement following ENDCASE, provided none of the executed statements caused transfer of control to another part of the program.
d. If a matching value is not found, the statement list after OTHERWISE is executed; if OTHERWISE is not present, the CASE statement has no effect, and execution continues with the statement following ENDCASE.

Note.

1. All <expression>'s in <case expression list> must be of the same type as <expression> in <case statement>.

FOR statement.

A statement to establish the initial, terminating, and incremental values of a control variable, which is used to determine the number of times a statement or a statement list contained in a loop is to be executed. The loop is repeated until the value of the control variable meets the termination condition or until a statement causes transfer of control from the loop.

Format

```
<for statement> ::=
   FOR <control variable> := <for list> DO <simple statement> |
   FOR <control variable> := <for list> DO
   <statement list>
   NEXT <control variable>
<control variable> ::= <simple numeric variable>
<for list> ::= <initial value> TO <final value> {STEP <step value>}
<initial value> ::= <arithmetic expression>
<final value> ::= <arithmetic expression>
<step value> ::= <arithmetic expression>
```

Statement execution

a. <initial value>, <final value> and <step value> are evaluated. If <step value> is not specified, it is assumed to be +1.
b. <control variable> is set equal to <initial value>.
c. If <step value> is positive (negative) and <control variable> is greater than (less than) <final value>, the termination condition is satisfied, and control passes to the first statement following the corresponding NEXT; otherwise step d is performed.
d1. The statement after DO is executed.
d2. The statement list after DO is executed.
e. <control variable> is set equal to
   <control variable> + <step value>
f. Step c is executed.

Notes.
---

1. After the execution of a FOR statement without transfer of control from the loop, the value of <control variable> is the first value satisfying the termination condition.
2. The <control variable> after NEXT is checked against the <control variable> after FOR; if they are not identical, an error occurs.

WHILE statement.

A statement to execute a statement or a statement list repetitively while the value of a logical expression is true.

Format

```
<while statement> ::=
            WHILE <logical expression> DO <simple statement> |
            WHILE <logical expression> DO
            <statement list>
            ENDWHILE
```

Statement execution.
---

a. <logical expression> is evaluated.
b. If the value of <logical expression> is false, the termination condition is satisfied and step e is performed.
c1. The statement after DO is executed.
c2. The statement list after DO is executed.
d. Step a is repeated.
e. Control passes to the first statement following the corresponding ENDWHILE, provided no statement caused transfer of control from the WHILE statement during step c2.

REPEAT statement.

A statement to execute a statement list repetitively until the value of a logical expression is true.

Format

```
<repeat statement> ::= REPEAT
                       <statement list>
                       UNTIL <logical expression>
```

Statement execution.

a. <statement list> is executed.
b. <logical expression> is evaluated, provided no statement caused transfer of control from the REPEAT statement during step a.
c. If the value of <logical expression> is false, step a is repeated.
d. If the value is true, the termination condition is satisfied and control passes to the first statement following UNTIL.

DATA statement.

A statement to provide values to be read to variables appearing in READ statements.

Format

```
<data statement> ::= DATA <value> { , <value>}
<value> ::= [+|-] <real number> | <string constant>
```

Notes.

1. The DATA statement is non-executable.
2. The values appearing in the DATA statement(s) form a single list. The first element in this list is the first value in the first DATA statement in the program. The last element in the list is the last value in the last DATA statement.

PROC - ENDPROC statement.

A statement to define a procedure or a function which can be called by means of an EXEC statement or in an arithmetic expression.

Format

```
<procedure declaration> ::=
                  <procedure head>
                  <statement list>
                  <ENDPROC part>
<procedure head> ::=
    PROC <procedure identifier> [ (<formal parameter list>) ]
<ENDPROC part> ::= ENDPROC <procedure identifier>
<procedure identifier> ::= <identifier>
<formal parameter list> ::=
      <formal parameter specification>
            { ,<formal parameter specification> }
<formal parameter specification> ::=
                   [ REF ] <simple variable> |
                   [ REF ] <simple string name> |
                   REF <numeric array name> ( [ ,] ) |
                   REF <string vector name> () |
```

Notes.

1. A procedure declaration specifies that <statement list> is treated as a unit named <procedure identifier>.
2. A procedure can be activated only by an EXEC statement or by a function call in an arithmetic expression. Return from the procedure occurs when an ENDPROC statement is executed.
3. Transfer of data between the calling program and the procedure or vice versa can be done using parameters. The transfer of data can also be done using global variables.
4. Formal parameters can be used in <statement list> as simple or subscripted variables, simple or subscripted string variables, or actual parameters.
   Formal parameters used as numeric array names or string vector names must specify the dimension of the array in the following way:

   () : one-dimensional array
   (,) : two-dimensional array

5. Whenever the procedure is activated, formal parameters in <statement list> will be assigned the values of (call by value) or replaced by (call by reference) corresponding actual parameters.
   Formal specification in the procedure head determines the choice based on the following rules:

<formal parameter spec.>	equal to
<simple variable>	call by value
REF <simple variable>	call by reference
<simple string name>	call by value
REF <simple string name>	call by reference
REF <numeric array name>	call by reference
REF <string vector name>	call by reference

   The REF before <numeric array name> and <string vector name> must be specified to allow for possible future extensions.
6. Activation of a procedure can take place as a function call in an arithmetic expression. A procedure used in this way should contain at least one assignment statement with the procedure identifier on the left side of ':='.
7. <procedure identifier> after ENDPROC must be the same as <procedure identifer> after PROC, otherwise an error will occur.
8. If the procedure was activated by an EXEC statement, execution of an ENDPROC statement will cause execution to continue with the statement after the EXEC statement.
   If the procedure was activated by a function call, the value of the function will be used in evaluation of the expression in which the call occurred.
9. Procedures may be called recursively.

Label statement.

A statement to define a label to which control can be transferred by a GOTO statement.

Format

<label statement> ::= <label name> :
<label name> ::= <identifier>

Note.
---

1. The label statement is non-executable.

DIM statement.

A statement to define storage for one or more numeric array variables or string variables.

Format

```
<dim statement> ::= DIM <declaration> {,<declaration>}
<declaration> ::= <numeric array declaration> ¦
                  <string variable declaration>
<numeric array declaration> ::=
 <numeric array name> ( <max row index> [,<max column index>])
<string variable declaration> ::=
 <simple string name> OF <string length> ¦
 <string vector name> (<max subscript>) OF <string length>
<max row index> ::= <simple numeric variable> ¦ <real number>
<max column index> ::= <simple numeric variable> ¦ <real number>
<max subscript> ::= <simple numeric variable> ¦ <real number>
<string length> ::= <simple numeric variable> ¦ <real number>
```

Statement execution
---

a. Storage space is allocated for the numeric array or the string variable for each declaration.

Notes.
---

1. A declaration of an array or a string must be executed before it is used in the program.
2. A numeric array or a string variable may be declared only once. Redimensioning of arrays or strings is not allowed.
3. If the value for <max row index>, <max column index>, <max subscript>, or <string length> does not evaluate to an integer, rounding is applied.
4. All of the elements in a declared numeric array are set to a value indicating undefined. A declared string variable is set to null ("").

Expressions.

Expressions are used in a number of different statements and constructions. An expression may be composed of parentheses, constants, variables (numeric or string), and functions, linked together by operators.

Format

```
<expression> ::= <logical expression> ¦
                 <arithmetic expression> ¦
                 <string expression>
```

Logical expressions.

A logical expression is used primarily for making the execution of a statement or a statement list conditional, but may also be used in assignment statements or as actual procedure parameters.

Format

```
<logical expression> ::= [NOT] <l-expression>
<l-expression> ::= <logical term> ¦
                   <l-expression> OR <logical term>
<logical term> ::= <logical operand> ¦
                   <logical term> AND <logical operand>
<logical operand> ::= <relation> ¦ (<logical expression>)
<relation> ::= <string relation> ¦ <arithmetic relation>
<string relation> ::=
    <string expression> <relational operator> <string expression> ¦
    <string expression> IN <string expression>
<arithmetic relation> ::=
     <arithmetic expression>
        [<relational operator> <arithmetic expression>]
<relational operator> ::= > ¦ >= ¦ = ¦ <> ¦ <= ¦ <
```

Notes.
---

1. The logical operators have the following meaning:

NOT:

A	NOT A
FALSE	TRUE
TRUE	FALSE

OR :

A	B	A OR B
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	TRUE

AND:

A	B	A AND B
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE

2. The relational operators have the following meaning:

```
>    : greater than
>=   : greater than or equal to
=    : equal to
<>   : not equal to
<=   : less than or equal to
<    : less than
```

3. If the relation between the two expressions is satisfied, the value of the relation is TRUE, otherwise FALSE.
4. If <relation> contains only an <arithmetic expression>, the relation has the value TRUE, if the value of the expression is not equal to zero, otherwise FALSE.
5. In a <string relation>, the two string expressions are compared character by character (from lower towards higher subscripts) on the basis of their ASCII decimal values. If a character in a given position in one string expression has a higher decimal value than the character in the corresponding position in the other string expression, the first string expression is the greater of the two. If the characters in corresponding positions are identical, but one string expression contains more characters than the other, the shorter string expression is the lesser of the two.

6. The IN operator (A$ IN B$) gives the index of the first occurrence of the first string expression in the second string expression. If the first string expression is not a substring in the second, the value of IN is 0. If the length of the first string expression is zero (LEN(A$) = 0), then IN = LEN(B$)+1.
7. The priorities of logical and relational operators are:

   First : Relational operators, IN
   Second : NOT
   Third : AND
   Fourth : OR

   When two operators have the same priority, evaluation proceeds strictly from left to right. Parentheses can be used to change the priority of logical and relational operators.

Arithmetic expression.

An arithmetic expression is a rule for computing a value of the numeric type. It is primarily used in assignment statements, but may also be used in logical expressions, PRINT statements, CASE statements and FOR statements.

Format

```
<arithmetic expression> ::= {<monadic operator>} <a-expression>
<monadic operator> ::= + | -
<a-expression> ::= <term> | <a-expression> + <term> |
                   <a-expression> - <term>
<term> ::= <factor> | <term> * <factor> |
           <term> / <factor> | <term> DIV <factor> |
           <term> MOD <factor>
<factor> ::= <operand> | <factor> ↑ <operand>
<operand> ::= (<arithmetic expression>) | <real number> |
              <numeric variable> | <system numeric function> |
              <numeric function> | (<logical expression>)
<numeric function> ::=
             <procedure identifier> [ (<actual parameter list>) ]
```

Notes.
---

1. The arithmetic operators have the following meaning:

+	:	monadic +	(+A)
-	:	monadic -	(-A)
↑	:	exponentiation	(A ↑ B)
*	:	multiplication	(A * B)
/	:	division	(A / B)
DIV	:	integer division	(A DIV B)
MOD	:	modulus calculation	(A MOD B)
+	:	addition	(A + B)
-	:	subtraction	(A - B)

   Exponentiation is standardized only for positive A.
   The result of an integer division (DIV) is standardized only for A >= 0 and B > 0
   The result of modulus calculation (MOD) is standardized only for A >= 0 and B > 0 (remainder).

2. During evaluation a floating point underflow may occur. In this case the result is set to zero. A floating point overflow will cause a run-time error.
3. A value must be assigned to a numeric variable before it may be used as an operand in an arithmetic expression. If this condition is not satisfied, a run-time error may occur.
4. The priorities of arithmetic operators are:

First	:	monadic + and -
Second	:	↑
Third	:	*, /, DIV, MOD
Fourth	:	+, -

   When two operators have the same priority, evaluation proceeds strictly from left to right. Parentheses may be used to change the priority of arithmetic operators.
5. If a logical expression is used as an operand, the value TRUE is equivalent to 1 and the value FALSE to 0.
6. In a call of a <numeric function>, actual parameters are treated in the same way as actual parameters in an EXEC statement.

System numeric functions.

System numeric functions may be used as operands in arithmetic expressions.

The following numeric functions exist:

ABS(X)	Absolute value of X.
ATN(X)	Arctangens of X, result in radians.
COS(X)	Cosine of X, where X is in radians.
EXP(X)	e to the power of X.
LOG(X)	Natural logarithm of X (X > 0)
SIN(X)	Sine of X, where X is in radians.
SQR(X)	Square root of X (X > 0).
TAN(X)	Tangens of X, where X is in radians.
INT(X)	Integer value of X. The largest integer less than or equal to X.
ORD(S$)	The ordinal number of the first character in S$.
LEN(S$)	The current length of S$

Note.

1. Identifiers must not be the same as names of numeric functions.

Numeric variables.

COMAL includes two types of numeric (real) variables: simple variables and subscipted variables.

Format

```
<numeric variable> ::= <simple variable> ¦
                       <subscripted variable>
<simple variable> ::= <identifier>
<subscripted variable> ::=
   <numeric array name> (<row index> [, <column index>] )
<numeric array name> ::= <identifier>
<row index> ::= <arithmetic expression>
<column index> ::= <arithmetic expression>
```

Notes.
---

1. A simple variable is referred to by using the identifier.
2. A simple variable may not be declared explicitly. The declaration is made automatically.
3. Subscripted variables are elements in arrays having either one or two dimensions.
4. Array variables must be declared in a DIM statement before they are used. Such a declaration contains the name of the array, its dimension, and the upper bounds for each index.
5. A subscripted variable must satisfy the following:
   1 <= row index <= upper bound for first index
   1 <= column index <= upper bound for second index
   If not, a run-time error occurs.
6. If the arithmetic expression for <row index> or <column index> does not evaluate to an integer, rounding is applied.
7. The value of a <numeric variable> is undefined before a value has been explicitly assigned to it.

String expression.

String expressions are used for assigning values to string variables (in assigment statements), for output (in PRINT statements), CASE statement and in relations.

Format

```
<string expression> ::= <string operand> { + <string operand>}
<string operand> ::= <string variable> | <string constant> |
                     <system string function>
<string constant> ::= "" | "<sequence of ASCII characters>"
```

Notes.
---

1. <string constant> can be empty or a sequence of characters, which may include letters, digits, spaces, and special characters except " and non-printable characters.
2. The string operator '+' denotes concatenation.

System string function.

System string functions may be used as operands in string expressions. The following system string function exists:

CHR$(X)	The ASCII character corresponding to the ordinal number X.

String variables.

COMAL 80 contains a type of variable called a string variable. The value of a string variable is a sequence of ASCII characters. There are two types of string variables: simple string variable and string array variable.

Format

```
<string variable> ::= <simple string variable> |
                      <subscripted string variable>
<simple string variable> ::=
          <simple string name> [ ( <selector> ) ]
<subscripted string variable> ::=
          <string vector name> ( <index> [, <selector> ] )
<simple string name> ::= <simple string identifier>$
<string vector name> ::= <string vector identifier>$
<index> ::= <arithmetic expression>
<selector> ::= <start position> [ : <substring length> ]
<start position> ::= <arithmetic expression>
<substring length> ::= <arithmetic expression>
<simple string identifier> ::= <identifier>
<string vector identifier> ::= <identifier>
```

Notes.
---

1. All string variables contain a dollar sign ($) after the identifier.
2. Substrings may be specified using a selector, containing the start position and length of the substring. If the length of the substring is not specified, it is assumed to be one.
3. All string variables must be declared in a DIM statement. For simple string variables, the maximum length of the string must be specified. For subscripted string variables, the number of strings and maximum length of a string must be given.
4 The following must be satisfied:
   1 <= subscript <= maximum number of strings
   1 <= start position <= string length
   1 <= start position + substring length - 1 <= string length
   0 <= substring length
   If not, a run-time error takes place.
5. If the arithmetic expression for <index>, <start position> or <substring length> does not evaluate to an integer, rounding is applied.
6. The value of a string variable is zero ("") before a value has been assigned to it. The value of a substring with a substring length equal to zero is zero ("").

Real numbers.

Real numbers may be used as operands in arithmetic expressions. Real numbers may be expressed as integers, decimal numbers, or in exponential form.

Format

```
<real number> ::= <decimal number> [<exponent part>]
<decimal number> ::= <integer> ¦ <integer>.<integer>
                     <integer>. ¦ .<integer>
<exponent part> ::=  E [+¦-] <integer>
<integer> ::= <digit>{<digit>}
<digit> ::= 0 ¦ 1 ¦ 2 ¦ 3 ¦ 4 ¦ 5 ¦ 6 ¦ 7 ¦ 8 ¦ 9
```

Notes.
---

1. A real number may not contain space characters.
2. The range of real numbers is not part of the standard.

Identifiers.

Identifiers are used to designate entities in a COMAL 80 program.

Format

<identifier> ::= <letter> {<letter> ¦ <digit> ¦ _ }
<letter> ::= < ASCII letters plus national characters >

Notes.
---

1. Identifiers may contain at least 16 characters.
2. Both capital and small letters may be used. No distinction is made between a capital and a small letter.
3. Space characters are not allowed within an identifier.
4. The character just before and after an identifier must neither be a letter nor a digit.
5. Identifiers must not be the same as the reserved keywords or names of system functions.
6. Identifiers designate entities in a program. The following types exist:
   - simple variables
   - subscripted variables
   - simple string variables
   - subscripted string variables
   - label names
   - procedure names
   - formal parameters.
7. An identifier may designate only one entity in a program.

Keywords.

COMAL 80 contains a number of keywords with a fixed meaning.

Format.

```
<keyword> ::=  AND ¦ CASE ¦ DATA ¦ DIM ¦ DIV ¦ DO ¦
               ELSE ¦ END ¦ ENDCASE ¦ ENDIF ¦ ENDPROC ¦
               ENDWHILE ¦ EXEC ¦ FOR ¦ GOTO ¦ IF ¦ IN ¦
               INPUT ¦ LET ¦ MOD ¦ NEXT ¦ NOT ¦ OF ¦ OR ¦
               OTHERWISE ¦ OUTPUT ¦ PRINT ¦ PROC ¦ READ ¦
               REF ¦ REPEAT ¦ RESTORE ¦ SELECT ¦ STEP ¦
               STOP ¦ TAB ¦ THEN ¦ TO ¦ UNTIL ¦ USING ¦
               WHEN ¦ WHILE
```

Notes.

1. Keywords may be typed using both capital and small letters.
2. A keyword must not be used as an identifier for other entities.
3. Space characters are not allowed in a keyword.
4. The character just before and after a keyword must neither be a letter nor a digit.