COMAL 80 PROGRAMMING LANGUAGE.

Preliminary proposal

Background.

The first COMAL (COMmon Algorithmic Language) was designed in 1974 by Boerge R. Christensen, the States Teacher Training College, Tonder, and Benedict Loefstedt, Depr. of Computer Science, University of Aarhus.

COMAL was constructed as an extension to BASIC, reflecting development in programming languages and techniques, such as structured programming.

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

Since 1974, the language has been extended with more facilities. Today, there exist several different versions of the language provided by a number og manufacturers.

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

The standardization should contain a kernel forming the COMAL 80 language and recommendations for extending the language.

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

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 supposed to operate in an interactive environment. COMAL statements are typed directly from a users terminal and immediately checked for syntactical errors.

Execution of programs are 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 description of the language is done in the 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 syntactical specification is made using the Backus-Naur Form extended with the following notation:

```
{ }  :  meaning zero or more occurencies
[ ]  :  meaning zero or one  occurencies.
```

Notes contains remarks concerning use of the construction, including rules, precautions, operation, and the like.

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

COMAL programs.

A COMAL program consists of a number of statements. Each program line begins with a line number. The rest of the line is made up of one or more COMAL words, with or without arguments. Each statement is written on one or more lines, depending on the statement type.

Format

```
<comal program> ::= <statement list>
<statement list> ::= { <comal statement> }
```

```
<comal statement> ::= <line number> <statement>
<line number> ::= <integer>
<statement> ::= <simple statement>|<structured statement>|
                <declaration statement>
<simple statement> ::= <assignment statement> | <io statement> |
                       <goto statement> | <return statement> |
                       <procedure call statement> |
                       <end statement> | <stop statement> |
                       <comment statement> |
                       <randomize statement>
<io statement> ::= <read statement> | <restore statement> |
                   <input statement> |
                   <print statement> | <print-using statement>
<structured statement> ::= <conditional statement> |
                           <repetitive statement>
<conditional statement> ::= <if statement> | <case statement>
<repetitive statement> ::= <for statement> | <while statement> |
                           <repeat statement>
<declaration statement> ::= <data declaration> |
                            <procedure declaration> |
                            <label statement> |
                            <on-err statement>
<data declaration> ::= <dim statement> | <data statement>
```

Notes.
---

1. <line number> is an integer, in the range 1 to 9999. The line numbers determines the order in which the statements are stored in the program.
2. <line number> must be followed by at least one space character.

Assigment statement.

A statement for assigning 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> | TAB |
```

<procedure identifier>
<string assignment> ::=
<string variable> := <string expression>

Statement execution.
---

a. the expression, string or numeric, is evaluated.
b. the result of step a is assigned to the variable on the left side of '='.
c. step a and b 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 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.

READ - statement.

A statement applied to read values from the list defined by one or more DATA statements and 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. The order in which variables appear in the READ statement is the order in which values for the variables are retrieved from the DATA statement list.
4. 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 situation occurs.
5. 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 situation occurs.
6. 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 [<comment>]

INPUT - statement.

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

Format

<input statement> ::=
  [<string expression>:] <variable> {,<variable>} [<continuation>]
<variable> ::= <numeric variable> | <string variable>
<continuation> ::= ;

Statement execution.
---

a. A prompt character is output, unless a string expression is included, in which case the value of the string expression is output.
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 pressing the 'end-of-input' key.
c. If <continuation> is not included, a carriage return (CR) and new line (NL) is output.

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 to a numeric variable can be typed in the following form:
   [+|-] <real number>
3. A separator between two real numbers may be any character not part of a real number. A separator between a real number and a string, is the first character not included in the real number.
4. If the data entered does not match the type of the current variable, an error reaction will take place, and the user can then enter data of the correct type.
5. If the data list is terminated before values have been assigned to all the variables in the argument list, a prompt string will be output, indicating that further items are expected.
7. Entering a value to a string variable followes the same rules as described in the assignment statement.

PRINT - statement.

A statement performing output of results at 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 determined by the value of a system defined variable, TAB.
2. If <print separator> is comma (,) between elements in the print list, the output of the last element will start 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 elment will always be printed on one line.
   If <print separator> is semicolon (;), the output of the next print element starts from the next character position. A blank space is always printed after a number.
3. If no <print end> is specified, a carriage return and line feed are output. The effect of a <print end> is the same as a <print separator> (note 2).
4. A PRINT statement with no <print list> causes a carriage return and line feed to be output.
5. TAB(exp) is a function to tabulate the printing position for an item in the print list to the column number evaluated from the 'exp'. Column number on the print line is numbered 1,2,,. If <arithmetic expression> evaluates to a column number greater than or equal to the current column number and less than the length of the print line, the value of 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 specied 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 is treated in the following way:
   # digit position and sign
   . decimal point (only if surrounded by #)
   All other characters in the format string is directly output.

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 being a part of a structured statement or a procedure declaration.
2. A goto statement transfering control out of one or more structured statements will terminate these.
3. The GOTO statement shall be used with much care, because, a program containing many GOTO statements is more difficult to debug. Also, the execution of a GOTO statement can be time consuming.

RETURN statement.

A statement to return from a procedure or a function.

Format

<return statement> ::= RETURN [<comment>]

Notes.
-----

1. If the procedure was activated by an EXEC statement, the execution will 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.
2. When the RETURN statement is executed, all structured statements started within the procedure must be terminated.

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 a RETURN ot ENDPROC statement is encountered. Then, the execution is continued with the statement immediately af the EXEC statement.

Notes.
---

1. If the procedure call contains an actual parameter list, 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 with actual parameters are the following:

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

An actual parameter being a <numeric array name> must have the same dimension 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 [<comment>]

Note.
---

1. An end statement will terminate execution of the program followed by a prompt output on the user's terminal.

Stop - statement.

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

Format

<stop statement> ::= STOP [<comment>]

Note
----

1. A stop statement will terminate execution of the program. A stop indication, including the line number of the stop statement, will be typed on the user's terminal. Then, control will be returned to interactive mode.

Comment - statement.

A statement for inserting explanatory comments within the program.

Format

<comment statement> ::= REM [<comment>]
<comment> ::= <sequence of ASCII graphic characters>

Notes.
-----

1. Comment statements are non-executable, but are stored with the program.
2. Optional text comments may also be inserted after the words RESTORE, RETURN, END, STOP, ENDIF, ELSE, ENDCASE, ENDWHILE, REPEAT, and ENDPROC.

RANDOMIZE - statement.

A statement to cause the random number generator to start at a different point in the sequence of random numbers generated by the RND(X) function.

Format

<randomize statement> ::= RANDOMIZE

Note.
---

1. Normally the same sequence of random number is generated by successive use of RND(X) function. If different starting points in the sequence are desired, the RANDOMIZE should be included in the program before the first occurrence of an RND(X) function.

Conditional 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>]
                <line number> ENDIF [ <comment> ]
<else part> ::= <line number> ELSE [ <comment> ]
                 <statement list>
```

Statement execution.
---

a. <logical expression> is evaluated.
b1. If the value is true, <simple statement> after THEN is executed, else the statement is skipped.
b2. If the value is true, <statement list> between THEN and ELSE/ENDIF is executed.
If <logical value> is false, <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 causes transfer of control to another part of the program, execution will continue at the first statement following ENDIF.

Note.
---

1. Line number for IF part, else part, and ENDIF part must form an increasing sequence.

CASE - statement.

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

Format

```
<case statement> ::= CASE <expression> OF
                     <case list element>
                     {<case list element>}
                     [<otherwise part>]
                     <line number> ENDCASE [ <comment> ]
<case list element> ::= <line number> WHEN <case expression list>
                     <statement list>
<case expression list> ::= <expression> {,<expression>}
<otherwise part> ::= <line number> 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 statements in the 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 statements after OTHERWISE are executed. If OTHERWISE is not present, the case statement is considered dummy.

Notes.

1. All <expression> in <case expression list> shall be the same type as <expression> in <case statement>.
2. Line numbers for the CASE part, the WHEN part, the OTHERWISE part, and the ENDCASE part must form an increasing sequence.

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 the 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 out of the loop.

Format

```
<for statement> ::=
   FOR <control variable> = <for list> DO <simple statement> |
   FOR <control variable> = <for list> DO
   <statement list>
   <line number> 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> after NEXT is checked against the <control variable> after FOR; if they are not identical, an error situation is encountered. If identical, the <control variable> is set to
   <control variable> + <step value>
f. Step c is executed.

Notes.

1. <step value> must not be zero.

2. After the execution of a for statement without transferring of control out of the loop, the value of <control variable> is the first value satisfying the termination condition.
3. The line number of the FOR part must be less than the line number of the NEXT part.

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>
            <line number> ENDWHILE [ <comment> ]
```

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 out of the WHILE statement during step c2.

Note.
---

1. The line number of the WHILE part must be less then the line number of the ENDWHILE part.

REPEAT - statement.

A statement to execute a statement list repetitively until the

value of a logical expression is true.

Format

<repeat statement> ::= REPEAT [<comment>]
<statement list>
<line number> UNTIL <logical expression>

Statement execution.
----

a. <statement list> is executed.
b. <logical expression> is evaluated, provided no statement caused transfer of control out of 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.

Note.
----

1. The line number of the REPEAT part must be less than the line number of the UNTIL part.

DATA - statement.

A statement providing values to be read into 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 a DATA statement or statements form a single list. The first element in this list is the first item in the lowest numbered DATA statement. The last element in the list is the last item in the

highest numbered DATA statement.

PROC - ENDPROC statement.

A statement for defining 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> [ CLOSED ]
                  <statement list>
                  <line number> <ENDPROC part>
<procedure head> ::=
    PROC <procedure identifier> [ (<formal parameter list>) ]
<ENDPROC part> ::= ENDPROC [ <comment> ]
<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 species that <statement list> shall be treated as a unit with the name, <identifier>.
2. A procedure can be activated by an EXEC statement or as a function call in an arithmetic expression. Return from the procedure occurs, when RETURN or ENDPROC statement is executed.
3. Transfer of data between the calling program and the procedure or vice versa can be done using parameters. If the procedure is not specified by 'CLOSED', it 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, actual parameters and procedure names. Formal parameters used as numeric array names or string vector names shall include specification of the dimension

of the array in the following way:

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

5. Formal parameters in <statement list> will, whenever the procedure is activated be assigned the values of (call by value) or replaced (call by reference) by 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

REF before <numeric array name> and <string vector name> shall be specified, because, of future possible extensions.

6. In a procedure specified as 'CLOSED', all simple variables except formal parameters, and labels are local for the procedure. Numeric arrays and string variables declared in DIM statements within a closed procedure are also local.
7. Activation of a procedure can take place as a function call in an arithmetic expression. A procedure used in this way must contain at least one assignment statement with the procedure identifier on the left side of '='.
8. Execution of ENDPROC has the same effect as the RETURN statement.

Label statement.

A statement for defining a label, so that control can be transferred to that point in the program by a GOTO statement.

Format

<label statement> ::= <label name> : [<comment>]
<label name> ::= <identifier>

Note.
1. The label statement is non-executable.

ON - ERR statement.

A statement to enable the programmer to take special action, if an error occurs during program execution.

Format

```
<on-err statement> ::= ON ERR THEN <simple statement>
```

Notes.

1. Usually a program is interruptet and an error message output, if an error occurs during program execution. If an ON - ERR statement has been executed, however, a run-time error will cause <simple statement> to be executed.
2. The type of error can be read by calling the system function SYS(X).

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> (<string length>) |
 <string vector name> (<max subscript>, <string length>)
<max row index> ::= <arithmetic expression>
<max column index> ::= <arithmetic expression>
<max subscript> ::= <arithmetic expression>
<string length> ::= <arithmetic expression>
```

Statement execution

a. For each declaration, storage space is allocated for the numeric array or the string variable.

Notes.

1. A numeric array or a string variable must be declared only once. Redimensioning of arrays or strings are not allowed.
2. A declaration of an array or a string shall be executed, before it is used in the program.
3. If arithmetic expression for <max row index>, <max column index>, <max subscript> or <string length> does not evaluate to an integer, rounding is applied.
4. All the elements in a declared numeric array is 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 paranthesis, 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 primary for making the execution of a statement or a group of statements 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>
<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 equal to
=    : equal to
<>   : not equal to
<=   : less than 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> only contains an <arithmetic expression>, the relation has the value TRUE, if the value of the expression is not equal to zero, otherwise FALSE.
5. In <string relation>, the two string expressions are

comparred 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 priority between logical and relational operators are:

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

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

Arithmetic expression.

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>)
              <string expression> IN <string expression>
<numeric function> ::=
             <procedure identifier> [ (<actual parameter list>) ]
```

Notes.

1. The arithmetic operators have the following meaning:

+	:	monadic +	(A + (+B))

```
  -   :  monadic -      (A + (-B))
  ¦   :  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 only defined for positive A.
The result of an integer division (DIV) is only defined for A >= 0 and B > 0

The result of modulus calculation (MOD) is only defined for A >= 0 and B > 0

2. During evaluation, a floating point underflow can 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 can be used as an operand in an arithmetic expression. If this condition is not satisfied, a run-time error occurs.
4. The priority of the arithmetic operators are:

```
First  :  monadic + and -
Second :  ¦ : exponentiation
Third  :  *, /, DIV, MOD
Fourth :  +, -
```

When two operators have the same priority, evaluation proceeds from left to right. Paranthesis can be used to change the priority of the arithmetic operators.

5. If a logical expression is used, the value TRUE is equivalent to 1 and the value FALSE is equivalent 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.
7. The IN operator gives the index 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.

System Numeric functions.

System numeric functions can be operand in arithmetic expressions.

Following numeric functions exist:

ABS(X)	Absolute value of X.
ATN(X)	Arctangent of X 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)	Tangent of X, where X is in radians.
INT(X)	Integer value of X.
RND(X)	Random number between 0 and X
SGN(X)	Algebraic sign of X.
ORD(S$)	The number of the first character in S$.
LEN(S$)	The current length of S$.
SYS(X)	System information based on the value of X.

Note.
---

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

Numeric variables.

COMAL includes two types of numeric (real) variables: simple variables and array 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 shall not be declared explicitly. The

declaration is done automatically, the first time a value is assigned to the variable.

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 upper bounds for each index.
5. A subscripted variable must satisfy the following relations:
   1 <= row index <= upper bound for first index
   1 <= column index <= upper bound for second index
   If not satisfied, a run-time error reaction occurs.
6. If arithmetic expression for <row index> or <column index> does not evaluate to an integer, rounding is applied.

String expression.

String expressions are used for assigning values to string variables (in LET statements), for output (in PRINT or INPUT statements), 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.

String variables.

COMAL 80 contains a type of variable called 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 (subscripted string variables).

Format

```
<string variable> ::= <simple string variable> |
                      <subscripted string variable>
<simple string variable> ::=
    <simple string name> [ ( <selector> ) ]
<subscripted stringvariable> ::=
          <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 can be used specifying a selector, containing a start position and the length of the substring.
3. All string variables must be declared in a DIM statement. For simple string variables, maximum length of the string must be specified. For subscripted string variables, the number of elements and length must be given.
4. Following relations must be satisfied:
   1 <= subscript <= maximum number of elements
   1 <= start position <= string length
   1 <= start position + substring length - 1 <= string length
   If not satisfied, a run-time error reaction takes place.
5. If arithmetic expression for <index>, <start position> or <substring length> does not evaluate to an integer, rounding is applied.

Real numbers.

Real numbers may be operands in arithmetic expressions. Number 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 up to at least 16 characters.
2. Both capital and lower case letters may be used. No differentiation is made between a capital and a lower case letter.
3. Space characters are not allowed in an identifier.
4. The character just before and after an identifier must not be a letter.
5. Identifiers must not be the same as the reserved keywords or name of system functions.
6. Identifiers designates entities in a program. Following types exist:
   - simple variables
   - subscripted variables
   - simple string variables
   - subscripted string variables
   - label names

- procedure names
- formal parameters.

7. One identifier must only designate one entity in the except for procedures specified as 'CLOSED'. The same rule shall be satisfied for each closed procedure.

Keywords.

COMAL 80 contains a number of keyword, having a fixed meaning.

Format.

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

Notes.
---

1. Keywords can be typed using both capital and small case letters.
2. A keyword must not be used as an identier for other entities.
3. Space characters are not allowed in a keyword.
4. The character just before and after a keyword must not be a letter.