Comal Kernel
Syntax & Semantics

DRAFT
As Amended by the
Comal80 Standardisation Group.
Copenhagen.

December 2-3 1983

Edited by : Kevin Ryan

1.0 INTRODUCTION

Each of the productions is numbered for reference. Superfluous or 1-productions have no semantics defined for them. Productions which just consist of a number of choices likewise have no semantics defined for them. We can assume that the semantics of A, where A ::= B | C | D is the semantics of B, C, or D, and that we have a parser which can make the correct choice.

2.0 CONVENTIONS

< >	Metalinguistic Brackets
::=	"is defined as"
[ ]	Optional
{ }	May occur zero or more times
|	Mutually exclusive alternatives

Note that the grammar is given mainly in iterative form, with the minimum of recursive definition.

Throughout this document "keywords" are shown in UPPER CASE.

WARNING

This is a draft standard and will be further revised. Some items currently under discussion are noted in the text, but prospective implementors of Comal80 are advised to contact the Group about any points not adequately covered by this document.

Write to:
Comal80 Standardisation Group,
Department of Computer Science,
Trinity College,
Dublin 2,
IRELAND

or phone: +353-1-772941 ext 1795/1765

3.0 PROGRAM STRUCTURE

1. <comal program> ::= <block>

2. <block> ::=
   {<declaration statement> |
   <non declaration statement>}

The semantics of a block is the semantics of each of its component statements, taken in the context left over by executing the previous statements.

3. <declaration statement> ::=
   <structured statement> |
   <unstructured declaration statement>

4. <non declaration statement> ::=
   <structured statement> |
   <unstructured statement>

5. <structured declaration statement> ::=
   <procedure declaration> |
   <function declaration>

6. <unstructured declaration statement> ::=
   <dim statement> |
   <data statement>

7. <structured statement> ::=
   <repetitive statement> |
   <conditional statement>

The structured Statements are composed of multiple lines. These lines are component lines. Each component line stands on its own as an input line and some syntax errors can be found. However further errors may be found when structures are checked, which can only be done when the program is complete.

8. <unstructured statement> ::=
<simple statement> <eol> |
<label statement> |
<remark> <newline>

9. <eol> ::=
[<remark>] <newline>

10. <remark> ::=
//{<displayable character>}

A remark consists of a [possibly empty] sequence of displayable characters, preceded by the sequence "//". The semantics of a remark is just its text with no interpretation by the system.

A comment can occur before any newline and will have no semantic effect on the preceding statements in the same line.

11. <newline> ::=
implementation dependent

12. <displayable character> ::=
implementation dependent

13. <simple statement> ::=
<stop statement> |
<return statement>|
<assignment statement>|
<input statement> |
<goto statement> |
<restore statement>|
<select statement>|
<open statement>|
<read statement>|
<write statement>|
<close statement>|
<delete statement>|
<print statement>|
<print using statement>|
<procedure call statement>|
<zone statement>

4.0 REPETITIVE STATEMENTS

14. <repetitive statement> ::=
    <while statement> |
    <repeat statement> |
    <for statement>

15. <conditional statement> ::=
    <if statement> |
    <case statement>

16. <while statement> ::=
    WHILE <logical expression> DO <eol>
      <statement list>
    ENDWHILE <eol>

The expression is evaluated. If it is false control passes to the next statement after the ENDWHILE, and the only effect is any side effects generated by evaluating the expression. If the expression evaluates to true, then the Statement list is executed. Control returns to the beginning of the WHILE statement, and the expression is re-evaluated. This pattern continues until the expression is found to be false. Control then passes to the statement following the "ENDWHILE".

17. <statement list> ::=
    {<non declaration statement>}

The semantics of a sequence of non-declaration statements is the semantics of each component non-declaration statement in the context of the machine state which results from executing all the previous component non-declaration statements.

18. <repeat statement> ::=
    REPEAT <eol>
      <statement list>
    UNTIL <logical expression> <eol>

The statement list is executed. The expression is evaluated. If it is false, control passes back to the first statement in the statement list. If the expression is true, then control passes to the next statement following UNTIL.

19. <for statement> ::=
        <short for statement> |
        <long for statement>

20. <short for statement> ::=
        FOR <for range> [<step>]
            DO <simple statement> <eol>

21. <long for statement> ::=
        FOR <for range> [<step>] DO <eol>
          <statement list>
        NEXT <control variable> <eol>

22. <for range> ::=
        <control variable> :=
          <initial value> TO <final value>

23. <step> ::=
        STEP <step value>

24. <control variable> ::=
        <numeric identifier>

25. <initial value> ::=
        <numeric expression>

26. <final value> ::=
        <numeric expression>

27. <step value> ::=
        <numeric expression>

The for initialisation is performed upon first entering the loop. This consists of evaluating the initial value and assigning the result to the control variable. Then the final value and the step value (if any) are evaluated. These three are evaluated once and for all and may not change during execution of the loop. The control variable is compared with the termination value. In the case of a positive step value, if the control variable is less than or equal to the termination value, then the statement list is executed. In the case of a negative step value, if the control variable is greater than or equal to the termination value then the statement list is executed. In all other cases the statement list is not executed and control passes to the statement following the for statement. After each execution of the statement list, the value of the control variable is incremented by the step value. If the step part is null, then the control variable is incremented by one.

The semantics of the short for statement is similar to the semantics of the long for statement, except that only a simple statement may occur after the "DO".

5.0 CONDITIONAL STATEMENTS

28. <if statement> ::=
        <short if statement> |
        <long if statement>

29. <short if statement> ::=
        IF <logical expression>
            THEN <simple statement> <eol>

30. <long if statement> ::=
        IF <logical expression> THEN <eol>
          <statement list>
        {ELIF <logical expression> THEN <eol>
          <statement list>}
        [ELSE <eol>
          <statement list>]
        ENDIF <eol>

First the logical expression immediately following the IF is evaluated. If the value of this expression is TRUE (non-zero), the statement list immediately following the THEN is executed, after which control passes to the statement following the ENDIF.

If the value of this expression is FALSE (zero), and there are ELIF sections, the logical expressions of each of the ELIF statements are evaluated in order. As soon as one of these evaluates are TRUE, the corresponding statement list is executed, and control then passes to the statement following the ENDIF.

If none of the logical expressions is true, and there is an ELSE statement present, the statement list following the ELSE is executed, after which control passes to the statement after ENDIF.

If none of the logical expressions is true, and there is no ELSE statement, control is passed to the statement after the ENDIF without any of statement lists being executed.

The Short If statement behaves in the same way as the Long If statement, except that only a simple statement may occur after the THEN.

31. <logical expression> ::=
        <numeric expression>

32. <case statement> ::=
        CASE <case selector> OF <eol>
        WHEN <choice list> <eol>
          <statement list>
        {WHEN <choice list> <eol>
          <statement list>}
        [OTHERWISE <eol>
          <statement list>]
        ENDCASE <eol>

33. <case selector> ::=
        <expression>

34. <choice list> ::=
        <numeric expression>
            {,<numeric expression>} |
        <string expression>
            {,<string expression>}

The case selector is evaluated. The choice lists are evaluated in the order in which they appear, until one is found with the same value as the case selector. When an identical value is found, then the statement list following the corresponding "WHEN" is executed. Control then passes to the statement following the ENDCASE. If none of the expressions in the "WHEN" choice lists match the "CASE" expression, the statement list following the "OTHERWISE" (if present) is executed. Control then passes to the next statement.

If none of the "WHEN" choice lists match the "CASE" selector, and no "OTHERWISE" clause exists, a runtime error occurs.

6.0 PROCEDURES AND FUNCTIONS

35. <procedure declaration> ::=
    PROC <procedure identifier> <head appendix> <eol>
      <procedure block>
    ENDPROC <procedure identifier> <eol>

36. <function declaration> ::=
    FUNC <function identifier> <head appendix> <eol>
      <function block>
    ENDFUNC <function identifier> <eol>

37. <function block> ::=
    <procedure block>

38. <procedure block> ::=
    {<import statement>}
    {<unstructured declaration statement> |
    <non-declaration statements>}

39. <head appendix> ::=
    [(<format parameter list>)][CLOSED]

40. <procedure identifier> ::=
    <identifier>

41. <function identifier> ::=
    <numeric identifier> |
    <string identifier>

42. <formal parameter list> ::=
    <formal parameter>
    {, <formal parameter>}

43. <formal parameter> ::=
    [REF] <variable identifier> |
    REF <variable identifier> <array indicator>

44. <import statement> ::=
IMPORT <variable identifier>
{, <variable identifier>} <eol>

45. <variable identifier> ::=
<numeric identifier> |
<string identifier>

46. <array indicator> ::=
[({,})]

Functions and Procedures differ only in the way they are invoked and in the way they return values. A procedure is called explicitly by an EXEC statement, whereas a function is called implicitly by the use of the function identifier in an expression. The effect of a function invocation is to provide a single returned value of the same type as the function identifier. Procedures can not return a value so procedure identifiers have no type (see also the RETURN statement).

The procedure or function can only be entered by calling it, and can only be exited by executing a RETURN statement or by control reaching the end of the procedure or function. The procedure or function identifier in the ENDPROC or ENDFUNC must match that in the heading. Parameters, other than arrays, are passed by value unless REF (pass by reference) is specified. Arrays are always passed by reference so REF must be included for them. Their number of dimensions are indicated by commas.

Procedures and functions are Open by default. Closed procedure or functions have there non-parameter variables allocated upon entre and these are lost when the procedure or function is exited.

Procedures or functions can be called recursively.

7.0 DIMENSION STATEMENT

47. <dim statement> ::=
    DIM <declaration>
    {,<declaration>} <eol>

48. <declaration> ::=
    <numeric declaration> |
    <string declaration>

49. <numeric declaration> ::=
    <numeric identifier> (<dimension part>)

50. <string declaration > ::=
    <string identifier> [(<dimension part>)] OF <length>

51. <dimension part> ::=
    <range> {,<range>}

52. <range> ::=
    [<lower bound>:] <upper bound>

53. <lower bound> ::=
    <numeric expression>

54. <upper bound> ::=
    <numeric expression>

55. <length> ::=
    <numeric expression>

There can be arbitrary numbers of dimensions. If no lower bound is specified a default of 1 is assumed. If the upper bound is less than the lower bound, an error message will be issued. Each element of the items declared in the DIM statement is initialised. Numeric items are set to

0, and string items to the empty string. If an attempt is made to redimension an existing array a runtime error occurs. This means that one cannot make an array bigger by redimensioning it.

The Standards Group believe that dynamic string handling is preferable. If dynamic strings are implemented the "OF <length>" part is optional. If the OF is present it sets the maximum length for the string.

56. <data statement> ::=
DATA <value> {,<value>} <eol>

All DATA statements in the program text are treated as a single data stream and are read in the order in which they occur in the program text.

For good programming practice the data statement should only appear immediately before the end of the program or the end of a procedure.

57. <value> ::=
[<sign>] <integer> |
[<sign>] <real number> |
<string constant> |
TRUE |
FALSE

Because there is no boolean type the values TRUE and FALSE are in all respects equivalent to one and zero respectively.

58. <sign> ::=
+ | -

8.0 EXPRESSIONS

59. <expression> ::=
    <numeric expression> |
    <string expression>

60. <numeric expression> ::=
    [<numeric expression> OR]
    <logical term>

61. <logical term> ::=
    [<logical term> AND]
    <logical factor>

62. <logical factor> ::=
    [NOT] <relation>

63. <relation> ::=
    <string relation> |
    <arithmetic relation>

64. <string relation> ::=
    <string expression>
    <relational string operator>
    <string expression>

65. <relational string operator> ::=
    IN | <relational operator>

NOTE

The semantics of this language feature have not yet been defined by the Comal80 Standardisation Group.

66. <arithmetic relation> ::=
    <formula> [<relational operator> <formula>]

67. <relational operator> ::=

    < | <= | = | >= | > | <>

68. <formula> ::=

    [<sign>] <arithmetic expression>

69. <arithmetic expression> ::=

    [<arithmetic expression> <adding operator>] <term>

70. <adding operator> ::=
    + | -

71. <term> ::=
    [<term> <multiplying operator>]
    <factor>

72. <multiplying operator> ::=
    * | / | DIV | MOD

The four multiplicative operators have semantics as follows:

* and / perform the normal multiply and divide operations.

The DIV function is defined to take two arguments x and y and to return the next lowest integer less than or equal to the result of dividing x by y.

The MOD function also takes two arguments x and y (y must be positive) and returns (x- (x DIV y)*y).

These definitions of MOD and DIV are such as to give the following results:

```
 36 MOD 5 is 1                36 MOD -5 is undefined
-36 MOD 5 is 4               -36 MOD -5 is undefined
 35 MOD 5 is 0                35 MOD -5 is undefined
-35 MOD 5 is 0               -35 MOD -5 is undefined

 36 DIV 5 is 7                36 DIV -5 is -8
-36 DIV 5 is -8              -36 DIV -5 is 7
 35 DIV 5 is 7                35 DIV -5 is -7
-35 DIV 5 is -7              -35 DIV -5 is 7
```

73. <factor> ::=
        <operand> [^<factor>]

74. <operand> ::=
        (<numeric expression>) |
        <constant> |
        <numeric variable> |
        <numeric function call>

75. <constant> ::=
        <integer> | <real number> |
        TRUE | FALSE

76. <real number> ::=
        <decimal number> [<exponent>]

77. <decimal number> ::=
        <integer>[.[<integer>]] |
        .<integer>

78. <exponent> ::=
        E[<sign>]<integer>

79. <integer> ::=
        <digit> {<digit>}

The allowable range of real and integer values is implementation dependent.

80. <numeric variable> ::=
        <numeric identifier> [(<subscript list>)]

81. <numeric identifier> ::=
        <real identifier>

82. <real identifier> ::=
        <identifier>

83. <subscript list> ::=
        <subscript> {,<subscript>}

84. <subscript> ::=
        <numeric expression>

Rounding: If a real value is used as a subscript it is first rounded to the nearest integer value.

85. <numeric function call> ::=
        <numeric identifier> [<actual parameter list>]

86. <string expressiuon> ::=
        <string operand> {+<string operand>}

87. <string operand> ::=
        <string constant> |
        <string variable> |
        <string function call>

88. <string constant> ::=
        "{<displayable character>}"

89. <string variable> ::=
        <string identifier>
        [(<subscript list>)]
        [(<substring specifier>)]

90. <string identifier> ::=
        <identifier>$

91. <substring specifier> ::=
    <from>:<to>

92. <from> ::=
    <numeric expression>

93. <to> ::=
    <numeric expression>

94. <string function call> ::=
    <string identifier>
    [(<actual parameter list>)]
    [(<substring specifier>)]

95. <stop statement> ::=
    STOP

The Program is suspended and control returns to the COMAL system. All variables retain their current values, and the program may later be restarted at the statement immediately following the STOP.

96. <return statement> ::=
    RETURN [<expression>]

Return causes the current procedure or function to be exited. A return with an expression is only allowed within a function. A return of any sort may NOT be used within the main program.

97. <assignment statement> ::=
    <assignment> {;<assignment>}

98. <assignment> ::=
    <numeric assignment> |
    <string assignment>

99. <numeric assignment> ::=
    <numeric variable> := <numeric expression>

100. <string assignment> ::=
<string variable> := <string expression>

In an expression containing both real and integer values the integers are "promoted" to reals and the expression evaluates to a real.

Precedence of Operators

The precedence of the various operators in Comal is shown in the following table. 1 is the highest evaluated left to right, except for exponentiation which evaluates from right to left.

1. Exponentiation	^
2. Multiplying Operators	* | / | DIV | MOD
3. Adding Operators	+ | - (Including Unary Minus)
4. Relational Operators	= | <> | < | > | <= | >= | IN
5. Logical Negation	NOT
6. Logical ANDing	AND
7. Logical ORing	OR

If an operand is an expression within parentheses, the value of the operand is the value of that bracketed expression. It follows that the order of evaluation can be modified by inserting matching pairs of parentheses.

9.0 INPUT STATEMENT

101. <input statement> ::=
INPUT [<string constant>:] <variable list> <print end> |
INPUT <file designator>: <variable list>

102. <variable list> ::=
<variable> {,<variable>}

103. <variable> ::=
<numeric variable> |
<string variable>

104. <file designator> ::=
FILE <channel number> [,<record number>]

105. <channel number> ::=
<numeric expression>

106. <record number> ::=
<numeric expression>

If the file designator is absent the variable list may be preceded by a prompt. If the prompt is null, then a system standard prompt such as "?" will be supplied. The system waits for the user to input a value. If the user makes a mistake and types in a value of the wrong type, the system prints an error message and re-issues the prompt. The user signals that he is finished typing in the values by sending a carriage return. Any number of values may be input on the same line, provided they are separated by commas.
If the file designator is present then no prompt is allowed as the input is read from an ASCII file. A file which is read by means of an INPUT statement is ASCII, whereas a file read by means of READ is binary.

10.0 GOTO STATEMENT

107. <goto statement> ::=
GOTO <label identifier>

The GOTO statement causes transfer of control to the label statements with the corresponding label identifier. The GOTO is restricted in where it can jump to.

1. A GOTO cannot transfer control into a structured statement. One can have a label statement within a structured statement, but only statements inside the structured statement may GOTO it.

2. A GOTO cannot transfer control into or out of a procedure.

3. Subject to 1. & 2. above, a GOTO may transfer control out of a structured statement.

108. <restore statement> ::=
RESTORE [<label identifier>]

The pointer to the next data item is reset to point to the first item in the (textually) first DATA statement. If the label is included the pointer is restored only to the DATA statement immediately following that label.

109. <label statement> ::=
<label identifier>:<eol>

The label statement is included as an unstructured statement. It may appear within a structured statement, although it only has scope (i.e. can only be jumped to) within the structured statement. Note that one may not include a label statement on the same line as another statement.

110. <label identifier> ::=
<identifier>

11.0 FILE HANDLING

111. <select statement> ::=
SELECT <type> <device specifier>
[,<dev info>]

112. <type> ::=
OUTPUT

113. <device specifier> ::=
<string expression>

114. <open statement> ::=
OPEN FILE <channel number>,<file name>
[,<dev info>],<mode>

115. <dev info> ::=
implementation dependent device information

116. <file name> ::=
<string expression>

117. <mode> ::=
READ |
WRITE |
APPEND |
RANDOM <record length>

118. <record length> ::=
<numeric expression>

If mode is random all records must be of the length specified. The string expression should evaluate to a file name. If the file is opened for write, and does not exist, then the system will create it. If the file is opened for read and does not exist, then a runtime error will occur.

If no error has occurred, the file will be associated with the channel number given.

119. <read statement> ::=
READ <variable list> |
READ <file designator>: <variable list>

120. <write statement> ::=
WRITE <file designator>: <variable list>

A record number in the file designator is valid only if the file is opened in random access mode. Otherwise the presence of a record number causes an error to occur. The expressions are written out in binary (if they are numeric) to the file opened on the given channel. If no file is attached to the channel, an error results. If the file is random access, then the combined bulk of all the data in the expression list should be less than or equal to the record size.

121. <delete statement> ::=
DELETE <file name> [,<dev info>]

If the named file is currently open a runtime error occurs. If it is closed it is deleted, and if it is reopened it will be a new file. It is permitted to delete a non-existing file, and no error results.

122. <close statement> ::=
CLOSE [FILE <channel number>]

The file attached to the channel specified by channel number is disconnected. If the file is open for sequential write, than an EOF is written before disconnecting. If the channel number given is not attached to any file, than an error occurs. If no channel number is specified than All currently open files are closed.

12.0 PRINT STATEMENT

123. <print statement> ::=
PRINT <output list> |
PRINT <file designator>: <output list>

124. <output list> ::=
[<print list> [<print end>]]

125. <print list> ::=
<print element>
{<print separator> <print element>}

126. <print element> ::=
<expression> |
<tab function>

127. <print end> ::=
<print separator>

128. <print separator> ::=
, | ;

129. <tab function> ::=
TAB (<numeric expression>)

If a file designator is specified, then the ASCII output goes out to the file rather than to the user's terminal. Formatted information may be written out to ASCII files.

Normally the expressions are printed out using a system-defined format for the particular type of expression.

130. <print using statement> ::=
PRINT USING <format info>:
<using list> [<print end>] |
PRINT <file designator>:
USING <format info>: <using list> [<print end>]

131. <using list> ::=
<using element> {,<using element>}

132. <using element> ::=
<numeric expression>

133. <format info> ::=
<string expression>

In the PRINT USING statement the format is determined from the string in the format clause. The meaning of the characters in the string of the Format Info is as follows: A substring of one or more embedded hash signs (#), optionally containing a single point (.). will be substituted on output by the value of a corresponding numeric expression. The expression will be printed out with as many digits precision as specified by hash signs to the right of the point. If no point is specified, no fractional part is printed. leading zeroes in the integer part are replaced by blanks; trailing zeroes in the fractional part are printed. If the expression has an integer part which is too large to be represented in the field, the field is printed out as all hash signs.

13.0 PROCEDURE CALL

134. <procedure call statement> ::=
[EXEC] <procedure identifier> [(<actual parameter list>)]

Syntactically, the only restriction on actual parameters is that they must be expressions. However, in order to send results back via a parameter passed by reference, the actual parameter must have a 1-value. (e.g. the 1-value of X is the address X, but the expression X+Y has no 1-value.)

A procedure may have side effects, both through parameters passed by reference and through shared variables.

SCOPE: Within a given scope, the first use of a name, whether as a procedure name, function name, label or variable name (whether DIMmed or not) defines the names type. Any subsequent attempt to redefine this name is an error.

BINDING:

1. Procedure names must be IMPORTed into CLOSED procedures.
2. OPEN procedures can NOT be IMPORTed.

ADVISORY NOTE

To ensure portability of software, users procedures should NOT rely on their formal parameters being available to called procedures.

135. <actual parameter list> ::=
<actual parameter> {,<actual parameter>}

136. <actual parameter> ::=
<expression>

137. <zone statement> ::=
ZONE ::= <numeric expression>

NOTE

The semantics of this language feature have not yet been defined by the Comal80 Standardisation Group.

138. <identifier> ::=
<letter> {,<letter> <digit> | _ }

If underscore is not available on the terminal to be used, another character may be chosen.

139. <letter> ::=
implementation dependent

140. <digit> ::=
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

14.0 APPENDIX A: STANDARD BUILT-IN FUNCTIONS

```
ABS(<numeric expression>) |
COS(<numeric expression>) |
SIN(<numeric expression>) |
TAN(<numeric expression>) |
ATN(<numeric expression>) |
LOG(<numeric expression>) |
EXP(<numeric expression>) |
SQR(<numeric expression>) |
INT(<numeric expression>) |
EOF(<numeric expression>) |
SGN(<numeric expression>) |
RND[(<numeric expression>)] |
RND(<numeric expression>) |
 ,(<numeric expression>) |
LEN(<string expression>) |
ORD(<string expression>) |
EOD | ZONE |
VAL(<string expression>) |
STR$ (<numeric expression>) |
CHR$ (<numeric expression>)
```

NOTE

The semantics of this language feature have not yet been defined by the Comal80 Standardisation Group.

15.0 APPENDIX B: STANDARD EXTENSIONS TO COMAL80

15.1 Integer Type Variables

To allow variables of type integer, replace the production:

```
<numeric identifier> ::=
     <real identifier>
```

with the two productions:

```
<numeric identifier> ::=
     <real identifier> |
     <integer identifier>

<integer identifier> ::=
      <identifier>£
```

Some flexibility between reals and integers is allowed. Integer numeric expressions may be assigned to real variables without any errors. Real expressions may not be assigned to integers, because this might result in truncation. The syntax of an integer expression is exactly the same as the syntax of an ordinary arithmetic expression, except that all components must be integers.

15.2 Short WHILE And REPEAT

One line versions of these statements are provided by modifying the productions for the Repetitive Statements.

```
<while statement> ::=
     <short while statement> |
     <long while statement>

<short while statement> ::=
     WHILE <logical expression>
             DO <simple statement> <eol>

<long while statement> ::=
     WHILE <logical expression> DO <eol>
       <statement list>
     ENDWHILE <eol>
```

```
<repeat statement> ::=
     <short repeat statement> |
     <long repeat statement>


<short repeat statement> ::=
     REPEAT
       <simple statement>
     UNTIL <logical expression> <eol>


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

The semantics of the short forms of the while and repeat are the same as those of the longer forms except that only a simple statement may be iterated.

15.3 STOP With A Message

instead of printing line numbers a STOP statement can result in a message being output. The syntac is:

```
<stop statement> ::=
     STOP [<string expression>]
```

15.4 Static Strings

If dynamic strings are NOT implemented a default string-length of 40 (forty) is given to an unDIMed string variable when it is first encountered..

INDEX

Assignment 18

Binding 26
Block 3
Built-in 28

Call 26
Case 9
Channel 20
Close 23
Closed 11, 26
Conditional 5
Conditional_statement 8

Date 13
Declaration 3, 12
Delete 23
Digit 27
Dim 12
Dimension_statement 12

Expression 14
Expressions 14

File 20
File_handling 22
For 6
Function 10
Functions 10

Goto 21
Goto_statement 21

Identifier 27
IF 9
Import 11
Imported 36
Input 20
Input_statement..... 20
Integer 29

Label 21
Letter 27

Message 30
Mode 22

Open 11, 22, 26

Parameter 10
Parentheses 19
Portability 26

Precedence 19
Print 24
Print_statement 24
Procedure 10
Procedure_call 26
Procedures 10
Program_structure 3

Read 23
Record 20
Remark 4
Repeat 5, 29
Repetitive 5
Repetitive_statement 5
Restore 21
Return 18
Rounding 17

Scope 21, 26
Select 22
Short 29
Static 30
Stop 18, 30
String 19
String-length 30

Tab 24

Using 25

Variable 11

While 29
Write 23

Zone 26