Virtual BASIC

External Reference Specification

B. G. Kenner
G. E. Kreuscher

DISCLAIMER:

This document is an internal working paper only. It does not represent any commitment on the part of Control Data Corporation.

Control Data Corporation

Revision Record	
Revision	**Description**
A 23 Mar 84	Preliminary version
B 17 Apr 84	Incorporates changes arising from Project Review of March 27, 1984
C 25 Jun 84	Incorporates changes and corrections arising from implementation

Table of Contents

1.0 INTRODUCTION AND OBJECTIVES

This document describes the Virtual BASIC language. Virtual BASIC will be implemented on the CYBER 180 as a compiler and a runtime library. Virtual BASIC is intended to provide a language which is

(1) similar to popular micro-computer BASICS, and

(2) easy to use, especially for casual users.

Virtual BASIC is, in its nature, not an optimizable language. No attempt to optimize Virtual BASIC object code will be made. However, the language provides an escape to FORTRAN. Given the high quality object code available from CYBER 180 FORTRAN, this escape should provide adequate performance for any application likely to be written in Virtual BASIC.

Virtual BASIC will conform to the ANSI Standard for Minimal BASIC. It will not conform to the proposed ANSI Standard for (full) BASIC.

2.0 REFERENCES

2.0 REFERENCES

Sunnyvale Product Design and Advanced Systems Design, CYBER 180 System Interface Standard, Revision H, September 21, 1982, DCS Log ID 2196.

Warburton, Pete, "USERS [sic] GUIDE: SOFTWARE DEVELOPMENT METHODOLOGY (SDM)," 8/27/82.

---, American National Standard for Minimal BASIC (X3.60-1978), January 17, 1978.

---, Draft Proposed American National Standard for BASIC (X3J2/82-17), October 1, 1982.

3.0 FEATURE DESCRIPTION

This section constitutes the language specification for Virtual BASIC. It introduces the notation used to describe syntax and defines the lexical, syntactic, and semantic properties of the language.

3.1 NOTATION

The notation used to describe syntax in this document is a variant of BNF, specifically, the variant used in the Draft Proposed American National Standard for BASIC. The reader is assumed to be comfortable with at least one variant of BNF; the purpose of the present section is to explain the particulars of the variant used in this document.

Here is the grammar for BNF. It is written in BNF. In other words, what follows is a description of the formal aspects of a notation. The notation in which the description is written is the notation being described. This sounds forbidding but is not. Readers who are comfortable with BNF probably can learn all they need to know about the variant used in this document simply by glancing at the following grammar. Readers who are uncomfortable should read the paragraphs which follow the grammar; those paragraphs are an informal rendering of the grammar in everyday English.

```
grammar = rule*
rule = name "=" alternation
alternation = sequence ("/" sequence)*
sequence = repetition repetition*
repetition = primary ("?"  / "*")?
primary = name / quoted-string / "(" alternation ")"
```

This grammar consists of six rules. The first of these says that if you have some rules and you lay them end to end, you get a grammar. More precisely, it says that a sequence of zero or more objects which are, formally, rules is an object which is, formally, a grammar. Or, if you like, that any object which is, formally, a grammar can be decomposed into a sequence of zero or more objects which are, formally, rules. The word "formally" appears above so many times because it is so important. Form is all that we are here concerned with. By no means all of the objects which are, formally, grammars are useful, sensible, or even correct. BNF is a compact and precise notation for discussion of matters of form but, by

3.0 FEATURE DESCRIPTION
3.1 NOTATION

itself, it is inadequate for a complete specification of BNF, much less of BASIC.

The second rule of the grammar says that a rule is a sequence of three things: a name, an equal-sign, and an alternation. We do not yet know what names and alternations are, but we know all there is to know about equal-signs. Quoted strings appear in rules where exact matches are required. So we don't yet know much about rules, but we know this: every rule contains an equal-sign.

The third rule says that an alternation consists of a sequence followed by zero or more two-part things. These two-part things consist of a slash followed by a sequence. In other words, the third rule says that an alternation is a non-empty list of sequences separated by slashes.

The fourth rule says that a sequence consists of one or more repetitions.

The fifth says that a repetition is a primary optionally followed by either a question mark or an asterisk. Note the importance of the parentheses. If they had been omitted, this rule would have said that a repetition is either a primary followed by a question mark or else a string which consists of a stand-alone asterisk if it is not empty.

The sixth rule says that a primary is one of three things: a name, a quoted-string, or an alternation enclosed in parentheses.

The grammar contains two undefined terms: name and quoted-string. The undefined terms of a grammar are called its pseudo-terminals. Usually, pseudo-terminals refer to classes of objects whose form the reader of the grammar is expected to know by some extra-grammatical means. Essentially, the pseudo-terminals of a grammar are its axioms.

In this grammar, a name is an ordinary English word or a series of such words separated by hyphens. A quoted-string is a character string enclosed in quotes. Embedded quotes in quoted strings are indicated by two successive quotes; thus the quoted string which consists of a single quote is denoted by four successive quotes: """".

In summary,

an asterisk in a grammar indicates that zero or more instances of something are permitted,

---

3.0 FEATURE DESCRIPTION
3.1 NOTATION

---

a question mark indicates that something is optional,

slashes indicate that any of several things might appear,

juxtaposition in the grammar indicates juxtaposition in whatever the grammar describes, and

parentheses indicate grouping.

3.2 LEXICAL MATTERS

This section describes the lexical properties of Virtual BASIC. The tokens of the language are defined. The significance of blanks and of alphabetic case is discussed. In-program commentary is described.

3.2.1 TOKENS

The tokens of Virtual BASIC are its lexical atoms.

```
token = integer / real-number / quoted-string /
     unquoted-string / name / integer-name /
     real-name / string-name / beginning-of-line
```

No token may cross a line boundary. No token that is neither a quoted-string nor an unquoted-string may contain an embedded blank.

3.2.1.1 Integers

An integer is a decimal, hexadecimal, or octal constant.

```
integer = digit digit* /
     "&" "H" hex-digit hex-digit* /
     "&" "O"?  octal-digit octal-digit*
hex-digit = digit / "A" / "B" / "C" / "D" / "E" / "F"
digit = octal-digit / "8" / "9"
octal-digit = "0" / "1" / "2" / "3" /
     "4" / "5" / "6" / "7"
```

Integers may contain any number of leading zero digits. Their values cannot exceed 2^63-1 (about 9.2*10^18); if they do, a fatal diagnostic is issued. (The caret indicates exponentiation. X^Y means X raised to the power Y.)

3.2.1.2 Real-Numbers

A real-number is either a plain-real-number optionally followed by an exclamation point or a number sign or else an integer unconditionally followed by an exclamation point or a number sign. The value of a real-number is the ordinary decimal value of the plain-real-number or integer to which the exclamation point or number sign, if any, is appended.

```
real-number = plain-real-number ("!"  / "#")?  /
     integer ("!"  / "#")
```

A plain-real-number contains a decimal point, a decimal exponent, or both.

```
plain-real-number = integer ("."  integer?
     decimal-exponent?  / decimal-exponent) /
     "."  integer decimal-exponent?
decimal-exponent = "E" ("+" / "-")?  integer
```

Real-numbers may contain any number of digits. Their magnitudes must be less than 2^4095 (about 5.2*10^1232). Real-numbers with extremely small magnitudes will be treated as zero by the compiler; no error message will be given. Real-numbers with magnitudes greater than or equal to 2^4095 will be diagnosed as fatal errors.

3.2.1.3 Quoted-Strings

Quoted-strings are character strings enclosed in quotes. A quote embedded in a quoted-string is indicated by two successive quotes.

```
quoted-string = quote (non-quote / double-quote)* quote
double-quote = quote quote
```

The syntax of a quoted-string could be more compactly, if more obscurely, presented using quoted-strings:

```
quoted-string = """" (non-quote / """""")* """"
```

A non-quote is any character at all other than quote. The length of a quoted-string is limited only by the number of characters which can be contained on a single line.

(Apostrophes have nothing special to do with quoted-strings. Here is a quote: " . Here is an apostrophe: ' .)

3.2.1.4 Unquoted-Strings

An unquoted-string is a run of characters which begins and ends with an unquoted-string-char and which contains only unquoted-string-chars and blanks.

```
unquoted-string = unquoted-string-char ((
     unquoted-string-char / " ")*
     unquoted-string-char)?
```

Any character other than blank, comma, quote, or colon is an unquoted-string-char.

Unquoted-strings are used only in data-statements. The length of an unquoted-string is limited only by the number of characters in a line.

3.2.1.5 Names

A name is a letter followed by a run of letters, digits, and periods.

```
name = letter name-char*
name-char = letter / digit / "."
```

Names may be up to 31 characters long. Longer names are diagnosed; the error is fatal.

3.2.1.6 Typed Names

An integer-name is a name followed immediately by a percent sign. A real-name is a name followed immediately by either an exclamation point or a pound sign. A string-name is a name followed immediately by a dollar sign.

```
integer-name = name "%"
real-name = name ("!"  / "#")
string-name = name "$"
```

Integer-, real-, and string-names may be up to 31 characters long. Longer integer-, real-, and string-names are diagnosed;

the error is fatal.

Integer-, real-, and string-names are, as their names suggest, names with which a particular data type is associated. (Plain) names can be used to denote objects of any type and are always used to denote objects with which no type is associated.

3.2.1.7 Beginnings-of-Line

No formal definition of the beginning-of-line token will be given. The reader's intuitive understanding of the beginning of a coded line is assumed to be adequate for the expository purposes of this document. It should be noted, however, that beginning-of-line is _not_ a character.

3.2.2 BLANKS AND THE LONGEST SCAN RULE

Blanks contained in quoted- and unquoted-strings are significant. Anyplace else, they are insignificant, except where they serve as token separators. No token which is neither a quoted- nor an unquoted-string may contain a blank. Thus,

```
FOR I
```

is entokened as two names (FOR and I);

```
1.0 E10
```

is entokened as a real-number (1.0) and a name (E10).

The compiler recognizes the longest token which begins at a given starting position. Thus, 12 is interpreted as a single integer (twelve) rather than two (one and two).

3.2.3 ALPHABETIC CASE

Alphabetic case is significant in Virtual BASIC only within quoted- and unquoted-strings. The compiler will produce a source list which faithfully reproduces alphabetic case as it was in the input file; however, internally, lower-case alphabetics will be translated to upper except in quoted- and unquoted-strings. Thus, for example, any variable name which appears in a diagnostic will be rendered in upper-case, regardless of how it may have appeared in the input file.

In this document, upper-case letters are used uniformly in all contexts in which case is not significent. Thus, the keyword FOR is rendered simply as "FOR" in the syntax, not as the full set of eight equivalent spellings:

```
("F"/"f") ("O/"o") ("R"/"r")
```

3.2.4 COMMENTARY

Virtual BASIC programs may contain embedded commentary. This commentary is not properly part of the program and has no effect on its meaning.

```
comment = "'" character* /
     statement-boundary "REM" character*
statement-boundary = beginning-of-line line-number?  / ":"
```

The apostrophe, wherever it appears outside of a quoted-string, marks the end of the significant portion of a line. The name REM, when it appears as the first token after a statement-boundary, does the same.

3.3 LINES, LINE NUMBERS, AND STATEMENTS

(In order to discuss line-numbers, we must introduce some terminology. A _routine_ is either an internal routine or an external-routine. An _internal-routine_ is an internal-function or an internal-subroutine. An _external-routine_ is a main-program, an external-function, or an external-subroutine. External-routines may contain embedded internal routines; internal routines may contain neither external- nor internal routines.)

A Virtual BASIC program consists of a series of lines. A line contains at most 255 characters. It begins with an optional line-number and contains zero or more statements separated by colons. Within an external-routine, the first line-number, if any, must be greater than zero; each following line-number must be greater than all the preceding line-numbers.

3.0 FEATURE DESCRIPTION
3.3 LINES, LINE NUMBERS, AND STATEMENTS

Any routine which contains an explicit reference to a line-number must also contain the line-number. Any routine which contains a line-number must also contain all explicit references to it. Thus, code in an internal routine may refer explicitly only to those line-numbers which are defined in that internal routine; code in an external-routine which is in no internal routine may refer explicitly only to those line-numbers which are defined in that external-routine and in no internal routine. It is, in other words, illegal to enter or leave an internal routine by means of an explicit line-number reference.

A line-number is associated with every statement of a Virtual BASIC external-routine. The associated line-number is that of the line which contains the statement if that line is numbered; if not, it is the line-number of the (lexically) most recent numbered line, or zero if the statement precedes the first numbered line.

3.4 DATA

This section describes the types, constants, and variables of Virtual BASIC.

3.4.1 TYPES

There are three types in Virtual BASIC: integer, real, and string. Integer data are numbers with integral values. Integers are stored as 64-bit two's complement binary numbers. Real data are numbers whose fractional parts are not necessarily non-zero. Real data are stored as 64-bit floating-point binary numbers. Strings are character data. Strings are stored, one ASCII character per byte, in as many contiguous bytes as are required to hold their current values.

With one major exception, mixing of integer and real data is freely permitted. The exception is parameter passing. Only integer data may be passed to integer formal parameters of user-written routines; only real data may be passed to real formal parameters. Most Virtual BASIC library routines will accept either integer or real data for numeric formal parameters. For example, both SIN(1) and SIN(1.0) are legal; each returns the real result which is (approximately) the sine of the angle whose radian measure is one.

Virtual BASIC contains no double precision type. It does, however, contain certain vestiges of that type; these vestiges are designed to make migration of programs from microcomputer BASICs to Virtual BASIC easy. Among the vestiges are variable names whose last character is the number-sign (such variables are double-precision variables in some microcomputer BASICs; they are real, i.e., single-precision, variables in Virtual BASIC), the DEFDBL spelling of the defreal-statement, and the library functions CVD, CDBL, and MKD$, which perform various kinds of type conversions. The attempt is to make some syntax portable from micro BASICs to Virtual BASIC. Since the real data of Virtual BASIC are about as precise as the double-precision data of many micro BASICs, if the syntax is portable, the numeric algorithm probably will be portable too.

3.4.2 CONSTANTS

There are four kinds of constants in Virtual BASIC; they correspond to integer, real-number, quoted-string, and unquoted-string tokens. These tokens are described in section 3.2.1 above. As their names suggest, integer tokens are constants of type integer, real-numbers are constants of type real, and quoted- and unquoted-strings are constants of type string.

There are no named constants in Virtual BASIC.

3.4.3 VARIABLES

This section describes the scalars and arrays of Virtual BASIC and defines the syntactic object variable, which is a reference to an array element, a scalar, or a substring of an array-element or scalar.

3.4.3.1 Scalar Variables

A scalar variable is a single, named, typed datum which is subject to modification by the program. The values of integer scalars must be greater than or equal to -(2^63) and less than 2^63. The values of real scalars must be greater than (-2^4095) and less than 2^4095. String scalars may contain any number of characters between 0 and 65,535, inclusive.

3.4.3.2 Arrays

An array is a named collection of data of like type. Each array element is subject to modification by the program. The limits on the values of array lements are the same as those on scalar variables of like type.

Arrays in Virtual BASIC can expand and contract as the program executes. The number of dimensions of an array is fixed at the time the program is compiled (up to 255 dimensions are permitted); the sizes and lower and upper bounds of the dimensions may vary at execution time.

The lower bound of any dimension of an array must be less than or equal to the upper bound of that dimension. Both lower and and upper bounds must be of magnitude less than 2^32. The total size of an array is limited by the amount of space available in Virtual BASIC's runtime heap; in no case may an array have 2^29 elements. An attempt to create an array whose storage requirement exceeds the unused capacity of the runtime heap will cause a fatal exception.

3.4.3.3 Variable References

The syntactic object variable defines the form of a reference to a Virtual BASIC array element or scalar or to a substring of either of these.

A variable is either a numeric-variable or a string-variable.

```
variable = numeric-variable / string-variable
```

A numeric-variable is either a numeric-scalar or a numeric-array-element.

```
numeric-variable = numeric-array-element /
     numeric-scalar
numeric-array-element = numeric-array-name subscript
numeric-array-name = numeric-identifier
subscript = "(" numeric-expression
     ("," numeric-expression)* ")"
numeric-scalar = numeric-identifier
numeric-identifier = name / integer-name / real-name
```

A string-variable is a whole-string-variable or a substring.

```
string-variable = substring /
```

```
        whole-string-variable
    whole-string-variable = string-array-element /
        string-scalar
    string-array-element = string-array-name subscript
    string-array-name = string-identifier
    string-scalar = string-identifier
    string-identifier = string-name / name
```

Substrings may be specified in two ways.

```
    substring = mid-substring / colon-substring
```

MID$ is a kind of addressing function. It specifies a substring of its first parameter, which may be either a whole-string-variable or a non-trivial string-expression. If its first argument is a whole-string-variable, then the MID$ reference is a legal left-hand side for an assignment.

```
    mid-substring = "MID$" "(" whole-string-variable ","
        first ("," length)?  ")"
    length = numeric-expression
```

The optional length defaults to 1. The mid-substring MID$(v$,f,l) is exactly equivalent to the colon-substring v$(f:f+l-1).

A colon-substring specifies a substring in terms of first and last character positions.

```
    colon-substring = whole-string-variable "(" first
        ":" last ")"
    first = numeric-expression
    last = numeric-expression
```

Let v$ be any whole-string-variable. Number the characters of v$ 1 through n, where n is the length of v$. The colon-substring v$(f:k) consists of characters r through s inclusive of v$, where

```
  r = cint(min(max(f,1),n))
```

and

```
  s = cint(max(min(k,n),1))
```

(Cint is a function which rounds its argument to an integer. Max and min are functions which return the maximum and minimum, respectively, of their arguments.)

If r is greater than s, the substring referenced is an empty substring which precedes the rth character of v$ and, if the (r-1)st character of v$ exists, follows the (r-1)st character.

3.4.4 SUPPLIED VARIABLES

Virtual BASIC supplies two variables at run-time: DATE$ and TIME$. DATE$ and TIME$ are string variables that may be both read and written and are available globally.

```
string-supplied-variable = "DATE$" / "TIME$"
```

3.4.4.1 DATE$

DATE$ is a ten-character string-variable of the form "mm-dd-yyyy", where mm is a two-digit ordinal for the month, dd is a two-digit day of the month, and yyyy is a four-digit year. For example, if the value of DATE$ were "07-04-2076", we might be celebrating the tricentennial of the first American revolution.

If DATE$ is never set by the user, its value is the current date known to NOS/VE. If the user sets DATE$, its value can denote any day from 01-01-0000 through 12-31-9999. If DATE$ is not at its maximum value, it is advanced when the value of TIME$ passes "00:00:00".

DATE$ may be set by assigning to it the value of a string-expression having the form "mm-dd-yy", "mm/dd/yy", "mm-dd-yyyy", or "mm/dd/yyyy". If either the month or the day is represented by a single digit, a leading zero is assumed. If only two digits are used for the year, the first two digits are assumed to be "19". If an impossible date is expressed or if the string is improperly formatted, an exception is raised.

3.4.4.2 TIME$

TIME$ is an eight-character string variable of the form "hh:mm:ss", where hh is the hour in the range "00" through "23", mm is the minute in the range "00" through "59", and ss is the second in the range "00" through "59". If TIME$ is never set by the user, its value is the current time known to NOS/VE. Once the user sets the value of TIME$, its current value is always the time to which it was last set added to the time elapsed since it was last set.

The value of TIME$ may be set by assigning to it the value of a string of the form "hh", "hh:mm", or "hh:mm:ss", where hh, mm, and ss are as above. If mm or ss is not specified, the value "00" is assumed. If a single digit is specified for any of hh, mm, or ss, a leading zero is assumed. An exception is raised if any of hh, mm, or ss are out of range or if the value of the string is improperly formatted.

3.5 CODE

This section describes the instructions according to which a Virtual BASIC program manipulates its data.

3.5.1 EXPRESSIONS

An expression is a numeric-expression or a string-expression.

```
expression = numeric-expression / string-expression
```

3.5.1.1 Numeric-Expressions

This section specifies the result types of the various operators used in numeric-expressions, describes the type conversions to which their operands may be subjected, and defines the syntax of numeric-expressions.

3.5.1.1.1 RESULT TYPES

All the logical operators (AND, OR, and so forth) produce integer results. All the relational operators produce integer results. The MOD and integer division (\) operators produce integer results.

The division (/) and exponentiation (^) operators always produce real results.

All the reamining numeric operators produce real results if either of their operands is of type real and integer results if both their operands are of type integer.

3.5.1.1.2 OPERAND CONVERSIONS

The operands of the logical operators are never type-converted. These operators perform bit-by-bit operations on 64-bit operands; whether the operand is of type integer or of type real is of no consequence.

The real operands, if any, of the integer division (\) and MOD operators are always converted to integers.

The integer operands, if any, of the division (/) and exponentiation (^) operators are always converted to type real.

For all other operators, if the operands are of like type, no conversion is performed. If they are mixed, the integer operand is converted to type real.

3.5.1.1.3 ORDER OF EVALUATION

The precedence of operators is indicated in the BNF and in the surrounding prose of the following section. Except where parentheses dictate otherwise, operators of higher precedence are applied before operators of lower precedence. If precedences are equal, operators on the left are applied before operators on the right. Thus, for example,

```
-A+B+C
```

is equivalent to

```
((-A)+B)+C
```

3.5.1.1.4 SYNTAX

At the highest level, a numeric-expression is a bit-by-bit logical expression. In order of decreasing precedence, the logical operators are NOT, AND, OR, XOR, EQV, and IMP. The corresponding operations are defined in the following truth table.

p	q	NOTp	pANDq	pORq	pXORq	pEQVq	pIMPq
0	0	1	0	0	0	1	1
0	1	1	0	1	1	0	1
1	0	0	0	1	1	0	0
1	1	0	1	1	0	1	1

(A Virtual BASIC operator never may be followed immediately by another operator. This rule, combined with the relatively high precedence of the unary operators (NOT and unary minus), gives rise to a peculiar distinction between first and subsequent instances of syntactic constructs. Thus, for example, the first equivalence in a numeric-expression is a different syntactic object from subsequent equivalences. The difference is that the first equivalence may begin with a NOT operator; subsequent equivalences may not.)

```
numeric-expression = first-eqv ("IMP" subs-eqv)*
first-eqv = first-xor ("EQV" subs-xor)*
subs-eqv = subs-xor ("EQV" subs-xor)*
first-xor = first-or ("XOR" subs-or)*
subs-xor = subs-or ("XOR" subs-or)*
first-or = first-and ("OR" subs-and)*
subs-or = subs-and ("OR" subs-and)*
first-and = logical-not ("AND" logical-primary)*
subs-and = logical-primary ("AND" logical-primary)*
logical-not = "NOT"?  logical-primary
logical-primary = relational-expression /
     arithmetic-expression
```

A relational-expression is a comparison. All relational operators have the same precedence. The result of a relational expression is an integer; the value is -1 (i.e., all bits set) if the relationship obtains, 0 (i.e., all bits cleared) if it does not.

```
relational-expression = string-expression relation
     string-expression (relation arithmetic-expression)* /
     arithmetic-expression relation arithmetic-expression
     (relation arithmetic-expression)*
relation = "=" /
     "<" ">" / ">" "<" /
     ">" "=" / "=" ">" /
     "<" "=" / "=" "<" /
     ">" /
     "<"
```

An arithmetic-expression, for the most part, is constructed in the usual way and obeys the usual precedence rules. Exponentiation is indicated by a caret ("^"); integer division is indicated by a backslash ("\"). The precedence of integer division is less than that of multiplication and ordinary division, greater than that of addition and subtraction. Negation is somewhat unusual in that it has higher precedence than addition and subtraction. The result of x MOD y is the remainder left by division of x, rounded to the nearest integer, by y, rounded to the nearest integer. There is no unary plus operator.

```
arithmetic-expression = first-mod (("+" / "-")
     subs-mod)*
first-mod = first-sum ("MOD" subs-sum)*
subs-mod = subs-sum ("MOD" subs-sum)*
first-sum = first-id (("+" / "-") subs-id)*
subs-sum = subs-id (("+" / "-") subs-id)*
first-id = first-prod ("\" subs-prod)*
subs-id = subs-prod ("\" subs-prod)*
first-prod = negation (("*" / "/") exponentiation)*
subs-prod = exponentiation (("*" / "/")
     exponentiation)*
negation = "-"?  exponentiation
exponentiation = numeric-primary ("^" numeric-primary)*
```

A numeric-primary is a numeric-variable, a numeric-constant, a reference to a function whose result is numeric, a supplied-read-only-variable, or a numeric-expression enclosed in parentheses. Parentheses affect the order of evaluation of numeric-expressions in the usual way.

```
numeric-primary = numeric-variable /
     numeric-constant /
     numeric-expression-function-ref /
     numeric-function-ref /
     "(" numeric-expression ")"
numeric-constant = integer / real-number
```

A numeric-expression-function-ref is a reference to a numeric expression-function. A numeric-function-ref is a reference to a numeric function which is internal, external, or supplied.

```
numeric-expression-function-ref = numeric-identifier
     exp-fn-actual-param-list?
exp-fn-actual-param-list = "(" expression
     ("," expression)* ")"
numeric-function-ref = numeric-identifier
     actual-parameter-list?
```

Parameters are passed to numeric-expression-functions by value; they are passed to internal and external numeric functions by address. See sections 3.5.2.1.2 and 3.5.4 for details.

3.5.1.2 String-Expressions

A string-expression is a string-primary concatenated with zero or more string-primaries. Concatenation is indicated by a plus sign.

```
    string-expression = string-primary
         ("+" string-primary)*
```

A string-primary is a string-variable, a reference to a string-valued function, or a string-expression enclosed in parentheses.

```
    string-primary = string-variable / string-function-ref /
         string-expression-function-ref /
         "(" string-expression ")"
```

A string-expression-function-ref is a reference to a string expression-function. A string-function-ref is a reference to a string function which is internal-, external-, or supplied.

```
    string-expression-function-ref = string-identifier
         exp-fn-actual-param-list?
    string-function-ref = string-identifier
         actual-parameter-list?
```

Parameters are passed to string-expression-functions by value; they are passed to internal and external string functions by address. See sections 3.5.2.1.2 and 3.5.4 for details.

3.5.2 STATEMENTS

3.5.2.1 Declarative Statements

Some of the declarative statements of Virtual BASIC are parts of multi-statement structures. (The external function statement is one of these.) Some stand alone. This section describes the latter class.

```
    declarative-statement = common-statement /
         expression-function-definition /
         function-declaration-statement /
         option-base-statement /
         subroutine-declaration-statement /
         type-declaration-statement
```

All the statements described in this section are executable in this sense: if control reaches one of these statements, it passes to the following statement. There is no requirement that these statements precede the executable statements, and they need not be grouped together. These statements have effect whether or not they are "executed". For example,

```
IF 0 THEN OPTION BASE 1
```

sets the default lower bound for array dimensions just as surely as

```
OPTION BASE 1
```

does.

The statements described in this section are effective over the whole of their containing external-routine. They have no effect on other external-routines which happen to be contained in the same source-deck.

3.5.2.1.1 COMMON-STATEMENT

The common-statement identifies arrays and scalars which are shared, either among routines of the same program or between programs which invoke each other by means of chain-statements.

```
common-statement = "COMMON" common-list
common-list = common-list-item ("," common-list-item)*
common-list-item = identifier ("(" ")")?
```

A common-list-item which consists only of an identifier denotes the scalar of that name; one which includes parentheses denotes the array whose name the identifier is. Order is not important in the common-statement; "COMMON A(), B" is exactly equivalent to "COMMON B, A()". Both statements say that the array A and the scalar B are to be shared; neither says anything about any relationship between their addresses. (In fact, Virtual BASIC arrays and strings do not have fixed addresses -- they may be relocated from time to time as their sizes change.)

The common-statement is unrelated to the COMMON statement of FORTRAN. If a FORTRAN subroutine is to have access to any of the data of a Virtual BASIC program, those data must be passed as parameters of the callx-statement which invokes the FORTRAN subroutine.

3.5.2.1.2 EXPRESSION-FUNCTION-DEFINITIONS

An expression-function-definition provides the archetype of a family of closely related expressions.

```
expression-function-definition = "DEF" (
     numeric-expression-function /
     string-expression-function )
numeric-expression-function = numeric-identifier
     ex-fun-formal-parameter-list?  "=" numeric-expression
string-expression-function = string-identifier
     ex-fun-formal-parameter-list?  "=" string-expression
ex-fun-formal-parameter-list = "(" identifier
     ("," identifier) ")"
```

An expression-function may have at most 255 parameters. Expression-function parameters are passed by value. Real actuals may be passed to integer formals; integer actuals may be passed to real formals. Whole arrays may not be passed to expression functions.

Changes made to formal parameters of expression-functions by functions which the expression-functions call have no effect on the corresponding actual parameters of the expression function. For example,

```
DEF BUMP1(X) = BUMP(X)
FUNCTION BUMP(X)
  X = X + 1
  BUMP = X
END FUNCTION
A = 1
PRINT BUMP1(A)
PRINT A
```

would print "2" and "1" on successive lines of standard output.

3.5.2.1.3 FUNCTION-DECLARATION-STATEMENT

The function-declaration-statement declares names to be those of functions, either internal or external.

```
function-declaration-statement = "DECLARE"
     "EXTERNAL"?  "FUNCTION" function-name-list
function-name-list = function-name
     ("," function-name)*
```

If "EXTERNAL" appears, the function-names are declared to be those of external-functions; if not, they are declared to be those of internal-functions. (Note that the function-declaration-statement cannot be used to declare the names of expression-functions. Expression-functions must be defined before they are first used. It is sometimes necessary to use an internal function before it is defined (because of recursive calls); the function-declaration-statement allows that to be done. It also provides a means of distinguishing an external-function from an undefined internal-function.)

3.5.2.1.4 OPTION-BASE-STATEMENT

The option-base-statement defines the default lower bound to be used in dim-statements.

```
option-base-statement = "OPTION" "BASE" ("0"/"1")
```

Default lower bounds are limited to zero and one. In the absence of an option-base-statement, the default lower bound is zero.

The option-base-statement has effect over the entire containing external-routine. An external-routine may contain at most one option-base-statement. The option-base-statement must precede all dim-statements in its containing external-routine.

3.5.2.1.5 SUBROUTINE-DECLARATION-STATEMENT

A subroutine-declaration-statement declares one or more names to be those of external subroutines.

```
subroutine-declaration-statement = "DECLARE"
     "EXTERNAL" "SUB" subroutine-name-list
subroutine-name-list = subroutine-name
     ("," subroutine-name)*
```

A subroutine-declaration-statement which declares an external subroutine must precede the first call to that subroutine.

3.5.2.1.6 TYPE-DECLARATION-STATEMENT

The type-declaration statement establishes a connection between the first letter of a name and the type associated with that name.

```
type-declaration-statement = defint-statement /
```

```
        defreal-statement / defstr-statement
    defint-statement = "DEFINT" letter-list
    defreal-statement = ("DEFSNG" / "DEFDBL")
        letter-list
    defstr-statement = "DEFSTR" letter-list
    letter-list = letter-list-item (","
        letter-list-item)*
    letter-list-item = letter-range / letter
    letter-range = letter "-" letter
    letter = "A" / "B" / "C" / "D" / "E" /
        "F" / "G" / "H" / "I" / "J" / "K" /
        "L" / "M" / "N" / "O" / "P" / "Q" /
        "R" / "S" / "T" / "U" / "V" / "W" /
        "X" / "Y" / "Z"
```

The type associated with every letter of the alphabet is real by default. For any letter of the alphabet, the default can be confirmed by a defreal-statement or overridden by a defint- or defstr-statement. No letter of the alphabet may be referenced in more than one type-declaration-statement in an external-routine. (We say that a letter is referenced in a type-declaration-statement if it is, by itself, a letter-list-item in that statement or if it is greater than or equal to the first letter, and less than or equal to the last, of some letter-range in that statement.)

The type associated by default with names which begin with letters referenced in a defreal-, define-, or defstr-statement is real, integer, or string, respectively.

Type-declaration-statements in an external-function apply to the function-name and to its formal-parameters. Type-declaration-statements in an external-subroutine apply to the formal-parameters of the subroutine. Except for references to names in external-sub- and external-function-statements, no type-declaration-statement which follows a reference to a name may alter the default type associated with the letter with which that name begins. Thus,

```
EXTERNAL FUNCTION Q
DEFSTR Q
```

is legal, while

```
Q = "How now, Brown Cow?"
DEFSTR Q
```

is not.

3.5.2.2 Executable Statements

3.5.2.2.1 ASSIGNMENTS

There are two assignment statements in Virtual BASIC: the let-statement and the swap-statement.

```
    assignment-statement = let-statement /
         swap-statement
```

3.5.2.2.1.1 Let-Statement

A let-statement assigns an expression to a variable or supplied variable. The keyword LET is optional. (It is the only initial keyword that is.)

```
    let-statement = "LET"?  (numeric-assignment /
         string-assignment)
    numeric-assignment = numeric-variable "="
         numeric-expression
    string-assignment = string-left-hand-side "="
         string-expression
    string-left-hand-side =
         string-supplied-variable / string-variable
```

If the left-hand side of a numeric-assignment is an integer, the value of the right-hand side is rounded to an integral value before it is stored. Thus,

```
  A% = -3.5
```

and

```
  A% = -4
```

have exactly the same effect.

3.5.2.2.1.2 Swap-Statement

The swap-statement exchanges the values of two variable.

```
    swap-statement = "SWAP" (numeric-variable ","
         numeric-variable / string-variable ","
         string-variable)
```

The semantics of

```
  SWAP v1, v2
```

are exactly those of

```
temp = v1 : v1 = v2 : v2 = temp
```

where temp is a compiler-generated temporary. The type of temp is the type of v1. If either v1 or v2 is a string-variable, both must be string-variables.

3.5.2.2.2 CONTROL

A control-statement is one which alters, or might alter, the flow of control in a program.

```
control-statement = call-statement /
     callx-statement /
     chain-statement /
     end-statement /
     error-statement /
     exit-function-statement /
     exit-sub-statement /
     gosub-statement /
     goto-statement /
     on-error-statement /
     on-gosub-statement /
     on-goto-statement /
     resume-statement /
     return-statement /
     run-statement /
     stop-statement
```

The on-error- and resume-statements are intimately involved in Virtual BASIC's run-time error processing mechanism. See section 5.2 for an overview of that mechanism.

3.5.2.2.2.1 Call-Statement

The call-statement is used to call a Virtual BASIC subroutine.

```
call-statement = "CALL" subroutine-name
     actual-parameter-list?
```

If the called routine takes no parameters, the actual-parameter-list is omitted. If present, the actual-parameter-list consists of a list of actual-parameters seperated by commas and enclosed in parentheses.

```
actual-parameter-list = "(" actual-parameter
     ("," actual-parameter)* ")"
```

An actual-parameter is an expression or an actual-array.

```
actual-parameter = expression / actual-array
actual-array = identifier "(" ","* ")"
identifier = name / integer-name / real-name /
    string-name
```

With the exception of expression-functions, parameter passing in Virtual BASIC is by address. If the actual parameter is an array or a scalar, the address passed is that of the actual parameter. In all other cases, the address passed is that of a temporary into which the value of the actual parameter has been stored. The called routine is free to modify any of its formal parameters. Such modifications, if they are made to formal arrays or to formal scalars to which scalars have been passed, are effective in the calling routine. Modifications of formal scalars to which constants, array elements, substrings, or non-trivial expressions have been passed are without effect in the calling routine.

If the called routine executes a dim- or an erase-statement which effects a formal-array, that statement has effect on the corresponding actual-array in the calling routine. If the called routine executes a clear-statement, any actual-arrays passed it by the calling routine are erased.

3.5.2.2.2.2 Callx-Statement

The callx-statement is used to call a FORTRAN subroutine (or any subroutine which conforms to the FORTRAN calling sequence).

```
callx-statement = "CALLX" external-routine-name
    external-parameter-list?
external-routine-name = letter (letter / digit)*
external-parameter-list = "(" external-parameter
    ("," external-parameter)* ")"
external-parameter = expression / external-array
external-array = numeric-identifier "(" ","* ")"
```

An external-routine-name may not be more than seven characters long.

One may not pass a string array as an external-parameter; other than that, external-parameters are just like actual-parameters. In particular, parameters are passed in the same way. Thus, scalars and arrays are subject to modification by the called routine; constants, array elements, substrings, and non-trivial expressions are protected.

There is no mechanism by which the FORTRAN (or other external) routine may alter the dimension-bounds of an actual-array.

Virtual BASIC cannot share data with FORTRAN subroutines by means of the common-statement; all data to be shared must be passed as parameters.

3.5.2.2.2.3 Chain-Statement

The chain-statement allows one Virtual BASIC program to end itself at the same time that it invokes another.

```
chain-statement = "CHAIN" file-name
file-name = string-expression
```

The file-name identifies the file from which the new program is to come. An exception is raised if it is not the local file name of a Virtual BASIC object program.

When a chain-statement is executed, the files of the old program are left open for use by the new. Variables and arrays which are to be shared by the old and new programs must appear in common-statements in both programs.

3.5.2.2.2.4 End-Statement

Execution of an end-statement terminates the execution of the program.

```
end-statement = "END"
```

The end-statement is an ordinary executable statement. It has no special lexical or syntactic properties. It need not be the last statement in a main-program and it may appear any number of times in any external-routine, internal-function, or internal-subroutine. Execution of an end-statement closes any open files.

3.5.2.2.2.5 Error-Statement

The error-statement raises an exception.

```
error-statement = "ERROR" numeric-expression
```

The numeric-expression, rounded to an integer value, is the number of the exception to be raised. It is possible to raise an undefined exception via an error-statement. When an exception not known to Virtual BASIC is not cleared by a resume-statement, the diagnostic written to the standard error file indicates an unrecognized error.

3.5.2.2.2.6 Exit-Sub- and Exit-Function-Statements

The exit-function- and exit-sub-statements are used to return from functions and subroutines, respectively. The function or subroutine may be either internal or external.

```
exit-function-statement = "EXIT" "FUNCTION"
exit-sub-statement = "EXIT" "SUB"
```

If there is an uncleared exception at the time an exit-function- or exit-sub-statement is executed, an exception will be raised in the calling routine. The only way to clear an exception is with the resume-statement.

3.5.2.2.2.7 Gosub-Statement

The gosub-statement jumps to the statement which follows a specified line-number and, in a sense, remembers where it came from.

```
gosub-statement = "GOSUB" line-number
```

At the time the jump is executed, the location of the statement following the gosub-statement is saved on the gosub stack. See also the return-statement.

(Gosub stacks are local to invocations of routines. Each time a routine is invoked, it begins execution with an empty gosub stack. Each time a routine terminates, any entries remaining on its gosub stack are discarded.)

3.5.2.2.2.8 Goto-Statement

The goto-statement performs an unconditional branch to the first statement associated with a specified line-number.

```
goto-statement = "GOTO" line-number
```

The line-number may be in a block which does not contain the goto. It is legal, if sometimes ill-advised, to jump into a

FOR loop or a block IF.

3.5.2.2.2.9 On-Error-Statement

The on-error-statement establishes the line-number of the current exception handling code for the containing routine.

```
on-error-statement = "ON" "ERROR" "GOTO" line-number
```

If the line-number is zero, default exception handling is enabled. Otherwise, user-supplied exception handling replaces default exception handling. When an exception is raised while user-supplied exception handling is in effect, an automatic branch is made to the statement following the line-number specified in the most recently executed on-error-statement. Typically, though not necessarily, the code at the indicated line-number will deal with the exception and then execute a resume-statement.

At entry to any routine, default exception handling is in effect for that routine. If a nonfatal exception is raised under default handling, an informative error message will be written to the standard error file and the routine will resume execution at the point of interruption. If the error is fatal, an error message will be queued for possible later use and the routine will terminate. If the routine is the main-program, any queued error messages will be written to the standard error file and execution will end; otherwise, an exception will be raised at the site of the call in the calling routine.

Thus, exceptions propagate outward from called routines to calling routines until a routine with non-default exception handling is found or there are no more callers. A side-effect of this propagation of exceptions is error traceback -- the messages queued will identify lines and routines where exceptions were raised. If the exception eventually is cleared by means of a resume-statement, the queue will be purged; if not, it will be written to the standard error file, thus providing the usual sort of line-and-routine traceback of fatal errors.

3.5.2.2.2.10 On-Gosub-Statement

The on-gosub-statement is an indexed gosub-statement.

```
on-gosub-statement = "ON" numeric-expression
     "GOSUB" line-number ("," line-number)*
```

The numeric-expression is evaluated and rounded to an integer. Let i be the value of that integer. A GOSUB is executed to the ith line-number in the list of line-numbers. If i is less than 1 or greater than the number of line-numbers in the list, no GOSUB is executed; instead, the (lexically) following statement is executed.

3.5.2.2.2.11 On-Goto-Statement

The on-goto-statement is an indexed goto-statement.

```
on-goto-statement = "ON" numeric-expression
     "GOTO" line-number ("," line-number)*
```

The numeric-expression is evaluated and rounded to an integer. Let i be the value of that integer. A GOTO is executed to the ith line-number in the list of line-numbers. If i is less than 1 or greater than the number of line-numbers in the list, no GOTO is executed; instead, the (lexically) following statement is executed.

3.5.2.2.2.12 Resume-Statement

The resume-statement clears an exception, purges the error message queue, and transfers control. It comes in three forms:

```
resume-statement = "RESUME" (
     "NEXT" /
     line-number /
     "0"?  )
```

All three forms clear the exception. The first form returns control to the statement (lexically) following the one in which the exception was raised. The second transfers control to the statement following the specified line-number. The third returns control to (the beginning of) the statement whose execution raised the exception. Note the possibility for loops using the third form. Note too the possibility for unpleasant surprises using the first. For example, if statement 100 in

```
100 IF exp GOTO 200
110 REM Could get here via resume ...
```

raised an exception during evaluation of the expression exp, and if the first form of RESUME is used by the exception-handling code, then control will be transferred to line 110, not to line 200, even though the value of exp may have been non-zero.

An exception is raised if a resume-statement is executed when there is no unresolved exception.

3.5.2.2.2.13 Return-Statement

The return-statement undoes a gosub-statement. That is, it pops the address of a statement off the gosub stack and jumps to that statement.

```
return-statement = "RETURN"
```

3.5.2.2.2.14 Run-Statement

The run-statement initiates the task denoted by the value of its string-expression.

```
run-statement = "RUN" string-expression
```

The value of the run-statement's string-expression is passed to the NOS/VE System Command Language Interpreter to be processed just as though it had been read from the command file. If any error results from the execution of the value of the string-expression as a command, an exception is raised.

3.5.2.2.2.15 Stop-Statement

The stop-statement suspends execution of the program and transfers control to the debugger.

```
stop-statement = "STOP"
```

3.5.2.2.3 INPUT AND OUTPUT

Virtual BASIC has a number of io-statements.

```
io-statement = close-statement /
     field-statement /
     get-statement /
     input-statement /
     line-input-statement /
     lprint-statement /
     lprint-using-statement /
     lset-statement /
```

open-statement /
print-statement /
print-using-statement /
put-statement /
rset-statement /
width-statement /
write-statement /
beep-statement

The io-statements are augmented by a collection of io-related library functions. Both the statements and the functions are described in this section.

A Virtual BASIC program can receive data from and send data to NOS/VE local files. A local file may be on either a mass storage or an interactive device. The NOS/VE standard files $INPUT and $OUTPUT are always available to the program during execution. Other local files can be made available to the program by specification in open-statements.

I/O with files is either sequential or random. Virtual BASIC sequential I/O reads and writes coded records. Random I/O reads and writes fixed-length sequences of binary bytes. Whether a file is opened for sequential or random I/O is controlled by the io-mode specification of the open-statement.

Sequential I/O is, just as one might hope, strictly sequential. Records are read one at a time from the beginning of the file. There is no capability for backward or forward skipping. A rewind is effected only by closing and reopening the file. Unless a file is opened with an io-mode of append, sequential output overwrites all previously defined information in a file. If a file is opened with an io-mode of append, records are written after the last preexisting record. Every time a new record is written to a sequential file, the end of the new record coincides with the file's (logical) end of information.

Random I/O offers arbitrary positioning of data transfers to and from a file and the improved performance of binary I/O. These benefits come at the expense of more trouble on the part of the BASIC programmer. Random I/O generally requires more specifications in the BASIC source text and more a priori information about the nature of one's data than does sequential I/O. Random I/O is frequently inappropriate for data intended for interchange with processors other than Virtual BASIC. Random I/O cannot be used with terminal files.

The Virtual BASIC functions and statements CLOSE, EOF, INPUT, LINE INPUT, LOC, LPRINT, OPEN, PRINT, PRINT USING, and WRITE are used with sequential I/O. The functions and statements CLOSE, CVD, CVI, CVS, FIELD, GET, LOC, LSET, MKD$, MKI$, MKS$, OPEN, PUT, and RSET are used with random I/O.

3.5.2.2.3.1 Open-Statement

The open-statement makes a NOS/VE file available to a Virtual BASIC program for I/O, establishes the io-mode, sets the channel number for references to the file in subsequent I/O statements, and allocates a buffer associated with the file.

```
open-statement = "OPEN" (open1 / open2)
open1 = file-name ("FOR" io-mode)?  "AS"
     "#"?  numeric-expression
     ("LEN" "=" numeric-expression)?
open2 = string-expression "," "#"?
     numeric-expression "," file-name
     ("," numeric-expression)?
io-mode = "INPUT" / "OUTPUT" / "APPEND"
```

If specified, the io-mode of the open1 form of the open-statement specifies the mode of access for a sequential file. If the io-mode is defaulted, RANDOM access is assumed. For the open2 form of this statement, the first character of the string-expression specifies the access mode as follows:

I denotes sequential INPUT
O denotes sequential OUTPUT
R denotes RANDOM I/O

Any other character will raise an exception. An exception will result also from an io-mode that is not conformable with the NOS/VE access mode of a preexisting file. An attempt to open an interactive device as RANDOM will raise an exception.

A file created as a result of a Virtual BASIC open-statement with an io-mode of RANDOM is assigned the NOS/VE file organization attribute BYTE ADDRESSABLE. A file created by an open-statement with an io-mode of INPUT or OUTPUT is assigned the file organization SEQUENTIAL. A preexisting file to be opened as RANDOM may have a file organization of either BYTE ADDRESSABLE or SEQUENTIAL. A preexisting file to be opened as INPUT or OUTPUT must have a file organization of SEQUENTIAL. An exception is raised if the file organization of a preexisting file is not compatible with the io-mode.

The file-name specified on the open-statement is the NOS/VE "file reference" indicating the device or file to be opened. Device types and file connections are easily established in the NOS/VE System Command Language and readily inferred by Virtual BASIC.

The first numeric-expression denotes the channel number to be used for subsequent references to the file or device. The optional preceeding "#" is of no semantic consequence; it is allowed only for compatibility. The value of the expression rounded to the nearest integer must lie between 1 and 99, inclusive. An exception is raised if the value of the channel number is out of bounds or if it is the value of a currently open channel.

The second numeric-expression is optional for both forms. Its value, rounded to the nearest integer, establishes the record size in bytes. If this is defaulted for a preexisting sequential file, the file's record length is used. If this is defaulted for any other file, a record length of 128 is used. If a value less than one is specified, an exception is raised. If a value is specified greater than the record length of a preexisting file, an exception is raised.

Multiple concurrent opens of a device or file are possible, but each must use a distinct channel number. Some combinations of concurrent opens of a file may have troublesome side effects. It is the responsibility of the programmer to provide appropriate access attributes for files intended to be so used.

A file or device other than default input or output must be opened in order to be accessed by any statement requiring a channel number.

If the file is not attached and the io-mode is INPUT, an exception is raised. If the file is not attached and the io-mode is OUTPUT, APPEND, or RANDOM, a new file is created.

Opening a file with an io-mode of OUTPUT overwrites any preexisting information in the file. Where the intent is to add data to a file while preserving the existing data, the APPEND io-mode should be specified.

The Virtual BASIC statements:

```
100 OPEN "SESAME" FOR INPUT AS #42
```

and

```
100 OPEN "INSCRUTABLE", #42, "SESAME"
```

have the identical effect of opening a file named "SESAME" for

INPUT, and establishing the channel number 42 for subsequent access to the file.

3.5.2.2.3.2 Close-Statement

The close-statement terminates access to a file or device through the channel number assigned to it at the time it was opened.

```
close-statement = "CLOSE" file-number-list?
file-number-list = "#"?  numeric-expression
     ("," "#"?  numeric-expression)*
```

A numeric-expression used in a file-number-list of a close-statement, when rounded to an integer, must be the value of a channel established by a previously executed open-statement. If no file is open for this value, an exception is raised. If no numeric-expression is specified on the close-statement, all open channels other than default input and output are closed.

Once a close-statement is executed for a channel, the file or device is no longer available to the program through that channel number. Output pending for the channel is flushed from its buffer at the time of the close.

The Virtual BASIC statement:

```
100 CLOSE
```

closes all open channels other than the default input and output files or devices.

The Virtual BASIC statement:

```
200 CLOSE #2, 4, 6, #8
```

closes the files or devices denoted by the channel numbers 2, 4, 6, and 8.

3.5.2.2.3.3 EOF Function

The EOF function is used to detect an end-of-file status. A reference to the EOF function appears in a Virtual BASIC program as:

```
"EOF" "(" numeric-expression ")"
```

The numeric-expression of the EOF function denotes the channel number of the instance of open for which the inquiry is made. If the value of the expression, rounded to the nearest integer, is not that of an open channel, an exception is raised.

The value returned by the function is an integer. If the file open to the indicated channel has reached its end-of-file, the integer value -1 is returned. Otherwise, the integer value zero is returned. Interactive devices and files opened for OUTPUT or APPEND are always at end-of-file. A file opened as RANDOM is never at end-of-file.

The following Virtual BASIC program illustrates the use of the EOF function in a relational expression (see the section on expressions) to detect an attempt to read beyond a file's data.

```
100 OPEN "MIND" FOR INPUT AS #9
110 IF EOF( 9 ) THEN END
120 LINE INPUT #9, LINSTR$
      .
      .
      .
200 GOTO 110
```

By the use of the EOF function on line 110, this program avoids the exception that would result from reading beyond the data in file MIND.

3.5.2.2.3.4 Field-Statement

The field-statement establishes fields in a random file's buffer that can be used for data extraction and insertion.

```
field-statement = "FIELD" "#"?  numeric-expression ","
     field-list
field-list = field-item ("," field-item)*                  |
field-item = numeric-expression "AS"                       |
     whole-string-variable                                 |
```

The first numeric-expression in a field-statement denotes a channel number. The value of the expression, rounded to an integer, is the channel assigned by a previous open-statement. If no channel of the specified value is open, or if the open channel is not a random file, an exception is raised.

Each subsequent numeric-expression in a field-statement denotes the length in bytes of the associated string-variable.

Bytes in the buffer are allocated in the order specified on the field-statement. If the sum of the bytes allocated exceeds the record length that was determined at the time the file was opened, an exception is raised. There is no limit to the number of field-statements that can be executed for a given buffer. The user is free to create as many different formats to reference his buffer as is convenient, but should keep in mind that careless use of overlapping fields can lead to unintelligible results.

The NOS/VE numeric representations used by Virtual BASIC all require eight bytes to contain a numeric value. A string-variable defined in a field-statement that is meant to be used with numeric data should have a length of eight bytes.

No data are moved as a result of the execution of a field-statement. The field-statement defines string-variables that coincide with pieces, or fields, of a random buffer. These string-variables are used as arguments of the CVI, CVS, or CVD functions for extracting numeric data from a buffer, as parts of a string-expression for extracting string data from a buffer, or as destination strings in lset and rset-statements for insertion of data into a buffer. They may also be used as string-variables in their own right, though they can be, in some sense, unstable.

A string-variable that has been defined by the execution of a field-statement which is later used on the left-hand-side of a string let-statement, in the variable list of an input- or read-statement, or as a formal parameter loses its residence in a buffer. A preexisting string-variable that is redefined by appearing in a newly executed field-statement loses its former value.

The following Virtual BASIC program illustrates the use of fields in a random file's buffer:

```
100 DATA 50
110 READ N
120 OPEN "SECRET" AS #7 LEN=28
130 FIELD #7, 20 AS COUNTY$, 8 AS POP$
140 FOR I = 1 TO N
150 GET #7
160 POP% = CVI(POP$)
170 PRINT USING " & COUNTY HAS A POPULATION OF #######_.";
COUNTY$,POP%
180 NEXT I
190 END
```

The field-statement at line 130 defines two string-variables that reside in the buffer for the random file "SECRET" open to channel number 7. The first of these string-variables, COUNTY$, coincides with the first twenty bytes of the buffer, and the second, POP$, coincides with the next eight bytes. With every execution of the get-statement at line 150, the values of these string-variables are liable to change. Note that even though the value of POP$ is numeric, it can be referenced in the random buffer only as a string. To be used as a numeric value in the program, it is first converted by the CVI function.

3.5.2.2.3.5 CVI, CVS, and CVD Functions

The CVI, CVS, and CVD functions are used to interpret values of string variables as numeric. References to these functions in a Virtual BASIC program appear as:

```
"CVI" "(" string-variable ")"
"CVS" "(" string-variable ")"
"CVD" "(" string-variable ")"
```

A string variable that is specified in a field-statement may have a numeric value. Before the value is available to the program as a number, it must be converted by one of these functions. CVI converts the string to an integer. CVS converts the string to a real number. CVD is equivalent to CVS; it is included only for compatibility. The conversion effected by these functions is a conversion of the interpretation, not of the data. That is, the representation is not changed, but the type is.

An arithmetic exception results when an argument of one of these functions cannot be interpreted as a number of the appropriate type by NOS/VE. The numeric representations of integers and floating-point numbers are eight bytes in length. A string-variable specified as the argument of one of these functions must have a length of eight bytes.

3.5.2.2.3.6 Get-Statement

The get-statement fetches a byte sequence from a random file into the buffer established for it by an open-statement.

```
get-statement = "GET" "#"?  numeric-expression
     ("," numeric-expression)?
```

The first numeric-expression in a get statement denotes the channel number assigned to the device or file by a previously executed open-statement. The value of the expression is rounded to an integer. If the specified channel is not open to a random file, an exception is raised.

The second numeric-expression denotes the ordinal of the byte sequence to be read. The value is rounded to an integer. If no such data are available, an exception is raised. If no ordinal is specified, the next byte sequence after the last get or put operation for this channel is read.

The get-statement fills a random file's buffer for subsequent data extraction by references to the string-variables defined in field-statements.

3.5.2.2.3.7 Input-Statement

The input-statement reads coded data from a sequential device or file and assigns them to program variables.

```
input-statement = "INPUT" ";"?                              !
     ("#" numeric-expression ",")?                          !
     (string-constant (";"/","))?                           !
     variable-list                                          !
variable-list = variable-list-item                          !
     ("," variable-list-item)*                              !
variable-list-item = numeric-variable /                     !
     whole-string-variable                                  !
```

The optional numeric-expression of an input-statement denotes the channel number established for the file or device at the time it was opened. The value is rounded to the nearest integer. If a value is specified that does not correspond to an open channel, an exception is raised. If no channel is specified, or if the value of the numeric-expression rounds to zero, the program's default input file or device is assumed. If the channel is not open with an io-mode of INPUT, an exception is raised.

If the channel from which input is requested is open to an interactive device, the operating system supplies the string "? " as a prompt to indicate that data are expected. The optional string-constant of the input-statement may be used to replace this default prompt. The programmer often finds it helpful to provide a prompt that is meaningful ("ENTER ID NO.:") or empathetic ("PLEASE TELL ME WHAT'S WRONG") in the context of an inquiry. If the user-specified prompt string is followed by a semicolon, the system default prompt is appended to it. If it is followed by a comma, the system default prompt is suppressed. The maximum length used for a prompt is

31 characters. If the value of a user-specified prompt exceeds this length, only the first 31 charaters are used. If a prompt string is specified for a channel that is not open to an interactive device, the prompt string is ignored.

Program variables into which data are to be read are specified in the input-statement's variable-list. The input-statement causes the next record to be read into the channel's buffer. If the channel is open to something other than an interactive device and no record is available, an exception is raised. If the device is interactive, a prompt is issued and the system will wait for a reply.

The data of an input reply must be separated by commas. The number of data in the reply must equal the number of items in the input-statement's variable-list. Data which are to be assigned to numeric variables must be valid numeric representations; data which are to be assigned to string variables may contain any characters. Data may be quoted or unquoted in the input reply regardless of the types of the variables to which they are to be assigned. Data received by integer variables are rounded to integral values before they are stored.

Note that substrings are not permitted in variable-lists. Neither the read-statement nor the input-statement may be used directly to assign a value to a substring.

An unquoted-string in an input reply differs slightly from the unquoted-string described in the previous section on tokens. An input unquoted string is a maximal run of characters on a single line which contains no leading blanks (it may contain embedded blanks) and no commas. All other characters, including colons and apostrophes, are permissible. Any datum that begins with a quote is assumed to be a quoted string. Only a quoted string datum can contain commas or leading or trailing blanks. A quote embedded in a quoted string is specified by two adjacent quotes.

If an erroneous input reply is furnished from an interactive device, Virtual BASIC will attempt to recover. The prompt "ERROR IN INPUT REPLY, PLEASE RESPECIFY" will be issued, and the system will wait for the entire input reply to be reentered.

3.5.2.2.3.8 Line-Input-Statement

The line-input-statement reads an entire line from a device or file into a string-variable.

```
line-input-statement = "LINE" "INPUT" ";"?
     ("#" numeric-expression ",")?
     (string-expression ";")?
     whole-string-variable
```

The optional numeric-expression denotes the channel number from which the line is read. The optional string-expression is used to replace the system's default prompt if the channel is open to an interactive device. Both of these optional parameters behave exactly the same as those on the input-statement.

If the length of a line read by the execution of a line-input-statement exceeds the maximum string length, an exception is raised and no assignment occurs. Otherwise, the entire line is assigned to the whole-string-variable.

3.5.2.2.3.9 LOC Function

The LOC function returns the ordinal of the current record of an open file or device. A reference to the LOC function appears in a Virtual BASIC program as:

```
"LOC" "(" numeric-expression ")"
```

The numeric-expression used as the argument of the LOC function denotes the channel number of the instance of open for which the inquiry is made. If the value of the expression, rounded to the nearest integer, is not that of a currently open channel that was assigned by an open-statement, an exception is raised.

If the channel is open as RANDOM, the LOC function returns the ordinal of the byte sequence that has been read or written by the most recently executed get- or put-statement for the channel. If no byte sequence has been read or written, the value zero is returned.

If the channel is open for INPUT, the function returns the number of the line most recently read. If it is open for OUTPUT or APPEND, the function returns the number of the line most recently written since the channel was opened. If no line has been read or written since the channel was opened, the value 1 is returned.

3.5.2.2.3.10 Lprint- and Lprint-Using-Statements

The lprint- and lprint-using-statements write coded data on a local file named "PRINT".

```
    lprint-statement = "LPRINT" print-list?
    lprint-using-statement = "LPRINT" "USING"
         string-expression ";" print-using-list ";"?
```

The lprint- and lprint-using-statements function the same as the print- and print-using-statements except that data so written are written to a local file named "PRINT". These statements are included in Virtual BASIC only for compatibility.

3.5.2.2.3.11 Lset- and Rset-Statements

The lset- and rset-statements insert the value of a string-expression into a string-variable.

```
    lset-statement = "LSET" string-variable "="
         string-expression
    rset-statement = "RSET" string-name "="
         string-expression
```

For each of these statements, the value of the string-expression becomes the value of the string-variable. Lset and rset differ from normal string assignments in that the length and location of the destination string are preserved. If the value of the string-expression has a length that is less than the length of the string-variable, lset left-justifies and blank-fills the value and rset right-justifies and blank-fills the value. If the length of the value of the string-expression is longer, the value is truncated on the right before the assignment. If the length of the destination string is zero, no assignment occurs.

If the string-variable is a field (see the field-statement) of a random file's buffer, the value of the string-expression is inserted into the buffer. Numeric values must be interpreted as string values for insertion into random buffers by lset or rset-statements (see the MKI$, MKS$, and MKD$ functions). For the sense of a numeric representation to be preserved, a string-variable used as a numeric field must be eight bytes in length.

3.5.2.2.3.12 MKI$, MKS$, and MKD$ Functions

The MKI$, MKS$, and MKD$ functions are used to interpret numeric values as string values. References to these functions in a Virtual BASIC program appear as:

```
"MKI$" "(" numeric-expression ")"
"MKS$" "(" numeric-expression ")"
"MKD$" "(" numeric-expression ")"
```

To insert a datum into a random file's buffer, the datum is assigned to a string variable that resides in the buffer as a result of a field statement. If the value to be inserted is numeric, one of these functions must be used to cause the value to be treated as a string.

These functions do not convert the values of their arguments in any way; they only convert the way Virtual BASIC treats the values. Since the representations of the NOS/VE numeric types used by Virtual BASIC are eight bytes in length, each of these functions returns the internal representation of the value of its argument as an eight-byte string variable. The three distinct names for this single capability are provided for compatibility.

3.5.2.2.3.13 Print-Statement

The print-statement writes coded data to a sequential file or device.

```
print-statement = "PRINT" ("#" numeric-expression ",")?
     print-list?
print-list = print-list-item
     ((","/";")?  print-list-item)* (";"/",")?
print-list-item = format-function / expression
format-function = "SPC" "(" numeric-expression ")" /
     "TAB" "(" numeric-expression ")"
```

The optional numeric-expression of a print-statement denotes the channel to which the data are written. The value of the expression is rounded to the nearest integer. If no value is specified, or if a specified value rounds to zero, the default output channel is used. If a specified value is not that of a channel open with an io-mode of OUTPUT or APPEND, an exception is raised.

The print-list is a list of expressions the values of which are to be written to the file or device open to the indicated channel. The print-list-items may be separated by blanks, commas, or semicolons. If the print-list is not specified, the print-statement has the effect of writing a blank line to the device or file.

Virtual BASIC print lines are divided into 14-character print zones. The positioning of the values printed is controlled by the use of punctuation and format-functions in the print-list. A comma in the print-list positions the next value printed at the beginning of the next print zone, blank-filling any intervening print positions that otherwise would have been unspecified. A semicolon in the print-list positions the next value printed immediately after the last one. Any number of spaces between print-list-items functions exactly the same as a semicolon.

The format-function SPC may be used in a print-list to insert spaces (blank characters) into the printed line. The number of spaces inserted is CINT(numeric-expression) MOD w+1, where w is the page width of the device or file. If this value is negative, an exception is raised. If this value is greater than the number of print positions remaining on the current line, an end of line is generated, and the next value printed is positioned at the beginning of the next line.

The format-function TAB may be used in a print-list to set the print position of the next value printed. The left-most print position is numbered 1. The print position used is CINT(numeric-expression) MOD w+1, where w is the page width of the file or device. If this value is not greater than zero, an exception is raised. If the current line is already positioned beyond this value, an end of line is generated, and the print position is set to this value on the following line with the intervening positions blank-filled.

If a print-list ends with a comma, semicolon, or format-function, no end of line is generated, and the next print to the same channel will begin at the current print position of the same line. Otherwise, the end of a print-list generates an end of line. Partial lines generated for interactive devices are transmitted when the end of the print-list is encountered. Partial lines destined for files and other devices are not transmitted until an end of line is generated either by encountering an end of a print-list with no comma, semicolon, or format-function or by closing the channel.

When the length of a value to be printed is greater than the number of print positions left to fill in the current line but less than the page width of the device or file, an end of line is generated and the value is printed at the beginning of the following line. If the length of the value to be printed is greater than the page width, as much of it as will fit on the current line is printed, and it is continued on as many lines as necessary.

A printed numeric value is always preceded by a space if it is positive and by a minus sign if it is negative. A printed numeric value is always followed by a space unless it is the last item on a line. If it is the last item on a line, it may be followed by an end of line. Numeric expressions whose values are integers are printed as integers. Numeric-expressions whose values can be represented no fewer accurately in a seven (or fewer) digit fixed-point format than in an exponential format are printed in a fixed-point format. All other numeric-expressions are printed in exponential format.

3.5.2.2.3.14 Print-Using-Statement

The print-using-statement writes data to a sequential device or file according to the specified format string.

```
print-using-statement = "PRINT"
     ("#" numeric-expression ",")?
     "USING" string-expression ";"
     print-using-list
print-using-list = expression ((","/";")?  expression)*
     (","/";")?
```

The optional numeric-expression of a print-using-statement denotes the channel to which the data are printed and is processed exactly the same as on the print-statement.

Ending a print-using-list with a comma or semicolon surpresses the generation of a line feed and carriage return in the same way as on a print-list of a print-statement. It makes no semantic difference whether expressions of a print-using-list are separated by a comma or by a semicolon. The print-using-list must contain at least one expression. The values of the expressions in the list are formatted according to the string-expression following the "USING" keyword of the print-using-statement.

The string-expression following the "USING" keyword is required. Certain characters that may appear in the value of the string-expression have particular meaning for data formatting and are described below. Other characters that do not have significance as formatting characters are replicated to the file or device as literals. If the value of the string-expression is ill-formed for use as a format string, an exception is raised.

When the expression being printed is a string, the format in which the value is printed is controlled by the appearance of "!", "\ ... \", or "&" within the format string. All of these control the length of the field in which a string value is printed. The character "!" indicates that only the first character, if any, of the string-expression's value is to be printed. If the value of the string is null, a blank is printed. The appearance of two backslashes separated by zero or more blanks indicates that the field length is the number of characters from the first backslash through the second (the number of blanks plus two.) If the value of the string-expression is longer, the value printed is truncated on the right. If the value of the string-expression is shorter, the value is printed left-justified and blank-filled. The character "&" indicates that the value of the string-expression is to be printed in a field the length of which is exactly equal to the length of the value.

When the expression to be printed according to a format string is numeric, the format strings "#", ".", "+", "-", "**", "$$", "**$", ",", and "^...^" may be used to control the display of the value. The values printed are rounded to fit the fields if necessary. If a value overflows a specified field, the truncated value is printed with the character "%" immediately to its left.

The appearance of a number sign ("#") denotes a digit position to be filled. The number of digit positions in a numeric field is the count of the number signs in the format substring that controls the field. If the display of the value requires fewer positions than are specified, the field is blank-filled on the left. One decimal point may appear in any position within the string of number signs. Its position indicates the position of the decimal point in the printed field. If the decimal point is not in the left-most position of the field, at least one digit is printed to the left of the decimal point.

The representation of the sign of a numeric value is controlled by the characters "+" and "-". If a "+" appears as the left-most or right-most character of a numeric format field, the sign of the value is printed immediately to the left or right, respectively, of the value. If a "-" appears as the right-most character of a numeric format field, a minus sign is printed to the right of a negative value. If neither "+" nor "-" appears and the value is negative, a minus sign is printed immediately to the left of the value.

3.0 FEATURE DESCRIPTION
3.5.2.2.3.14 Print-Using-Statement

Beginning a numeric format field with two asterisks ("**") has the effect of specifying two additional digit positions for the printed field and of filling the leading blanks of the printed field with asterisks.

Beginning a numeric format field with two dollar signs ("$$") has the effect of specifying two additional digit positions for the printed field and causes the position immediately to the left of the left-most digit printed to be filled with a dollar sign. The only sign specification that is allowed with the dollar signs is the trailing minus variation described above. Exponential formats (see below) cannot be combined with dollar signs.

Beginning a numeric format field with the string "**$" combines the two previous specifications. This specifies three additional digit positions for the printed field. The position immediately to the left of the left-most digit printed is filled with a dollar sign, and any blank positions to its left are filled with asterisks.

A comma specified within the digit positions of a numeric format field has the effect of adding commas to the printed representation of the numeric value. A comma is inserted to the left of each third digit to the left of the decimal point if there is another significant digit to its left. A comma that appears at the beginning or end of a format field is replicated as a literal. A comma used in an exponential format has no effect.

Three or more carets ("^") may be placed after the digit position specification of a numeric format field to indicate an exponential format. Each caret specified indicates a character position of the exponent field. Three carets are sufficient to specify a minimal exponent field of the form E±n or D±n. If the exponent to be printed overflows the field specified, the value is printed with the character "%" immediately to the left.

Any of the characters enumerated above as having significance as format specifications may be used as a literal in a format string by preceding it by the underscore character. The literal character appears in the formatted line; the underscore does not.

3.5.2.2.3.15 Put-Statement

The put-statement writes the data in a random file's buffer to the file.

```
put-statement = "PUT" "#"?  numeric-expression
     ("," numeric-expression)?
```

The first numeric-expression of the put-statement is required. Its value rounded to an integer denotes the channel number for which the buffer is to be written. If no such channel is open, or if the channel is open with an io-mode other than RANDOM, an exception is raised.

The optional second numeric-expression of the put-statement,if present, denotes the position in the file at which the data are to be written. Its value rounded to the nearest integer must be within the range allowed by NOS/VE Access Methods for a file's record limit (type amt$record_limit). If the value is outside this range, an exception is raised and no data are written. If no number is specified, the data are written as the next byte sequence after that accessed by the last executed get or put operation for the same channel.

3.5.2.2.3.16 Width-Statement

The width-statement sets a page width for an output file.

```
width-statement = "WIDTH"                                        |
     (("#"? numeric-expression / file-name) ",")?                |
     numeric-expression
```

The optional numeric-expression of the width-statement denotes the channel number for which a page width is to be set. If its value rounded to to the nearest integer does not denote a channel open for sequential output, an exception is raised. A change made by a reference to the channel number is in effect only for the instance of open denoted by the channel number.

A page width may also be set for a file-name. A file-name specified on a width-statement is a NOS/VE "file-reference". An exception is raised if the user references a file over which his control is insufficient to allow a change of the page width attribute.

Setting the page width by file-name has no effect on any current open of a file or device indicated by a file-name, but is used for all subsequent opens. If neither a channel number nor a file-name is present in a width-statement, or if a channel number that rounds to zero is specified, then the current default output channel is assumed.

The value of the required numeric-expression is used for the new page width. If the value is less than fourteen (the length of a print zone), an exception is raised. Otherwise, the value used is CINT(numeric-expression) MOD 256. The page width determines the maximum number of characters that can be printed before a line feed and carriage return are generated.

3.5.2.2.3.17 Write-Statement

The write-statement writes lines of coded data to a sequential device or file.

```
write-statement = "WRITE" ("#" numeric-expression ",")?
     write-list?
write-list = expression ((","/";") expression)*
```

The optional numeric-expression of the write-statement denotes the channel to which the line is to be written. The value of the expression is rounded to the nearest integer. If no channel is specified, or if the value specified rounds to zero, the default output channel is used. If a value is specified that corresponds to no channel open with an io-mode of "OUTPUT" or "APPEND", an exception is raised.

The list of expressions is optional. If none is specified, only an end of line is transmitted to the channel. The value of each expression specified is formatted and transmitted in the manner of a print-statement's print-list that contains items all of which are separated by semicolons except that commas are printed to seperate the values, quotes are printed to delimit string values, quotes embedded in strings are doubled, and a positive numeric value is printed without a leading blank. An end of line is always transmitted to the channel after the values of the expression-list are written.

3.5.2.2.3.18 Beep-Statement

The beep-statement transmits the ASCII bell character to a terminal.

```
beep-statement = "BEEP"
```

The beep-statement is included in Virtual BASIC only for compatibility. It is equivalent to "PRINT CHR$(7);".

3.5.2.2.4 DATA INITIALIZATION

Uninitialized numeric-variables have the value zero; uninitialized string-variables are null. Variables may be initialized from an external-routine's data table. Values are established in an external-routine's data table by data-statements. Variables are initialized from an external-routine's data table by read-statements. A datum from a data table may be selected for a subsequent read-statement by a restore-statement.

```
data-initialization-statement = data-statement /
     read-statement / restore-statement
```

The data-initialization-statements of Virtual BASIC are described in more detail in the remainder of this section.

3.5.2.2.4.1 Data-Statement

Data statements are used to establish an ordered set of data for an external-routine. A data-statement contains a list of data separated by commas.

```
data-statement = "DATA" datum ("," datum)*
datum = unquoted-string / quoted-string
```

All the data-statements of an external-routine, taken together, define a single collection of data. From which data-statement a particular datum may have come is of no significance. The single data-statement

```
DATA 1, 2
```

is exactly equivalent to the pair

```
DATA 1 : DATA 2
```

The data in the collection defined by external-routine's data-statements are accessed by means of read-statements.

Data in data-statements are all treated as strings of characters. They need not be quoted unless they contain apostrophes, colons, commas, or significant leading or trailing blanks.

3.5.2.2.4.2 Read-Statement

The read-statement assigns values from the external-routine's data table to the variables of the statement's variable list.

```
read-statement = "READ" variable-list
```

The variable-list of a read-statement must be non-empty. The data to be read reside in the data table of the external-routine as a result of having appeared in data-statements. If the data remaining in the external-routine's data table are too few to satisfy the variable-list, an exception is raised. Values in the data table can be made available for reuse by the restore-statement.

The values of the data table are assigned to the variables of the read-statement's variable-list in sequential order. Any value may be assigned to a string; only data which are valid numeric representations may be assigned to numbers. Quoted data are treated no differently from unquoted data. Values assigned to integers are rounded to integral values before they are stored. If an attempt is made to assign a datum which is not a valid representation of a number to a numeric-variable, an exception is raised.

3.5.2.2.4.3 Restore-Statement

The restore-statement is used to reset the external-routine's pointer into its data table. See also the read-statement.

```
restore-statement = "RESTORE" line-number?
```

If the line-number is omitted, the pointer is reset to the beginning of the data table; the next READ will get the first constant on the first data-statement in the external-routine. If the line-number is present, the pointer is reset to the first constant of the first data-statement whose line-number is at least as large as the line-number specified by the restore-statement.

3.5.2.2.5 MISCELLANEOUS EXECUTABLE STATEMENTS

This section describes the executable statements which do not fit naturally into any of the groups previously described.

```
miscellaneous-executable-statement =
     clear-statement /
     dim-statement /
     erase-statement /
     randomize-statement
```

3.5.2.2.5.1 Clear-Statement

The clear-statement discards the data of the external-routine in which it appears.

```
clear-statement = "CLEAR"
```

Numeric scalars are set to zero; string scalars are set to the null string. Arrays, whether numeric or string, are erased. (See also the erase-statement.)

The clear-statement applies to all arrays known to the containing external-routine. In particular, it applies to arrays in common and to formal-arrays. If a called routine executes a clear-statement, any arrays it declares in common and any actual-arrays passed it by its caller are erased. The erasures are immediately effective in both the called and the calling routine.

3.5.2.2.5.2 Dim-Statement

The dimensions of arrays are established and changed by means of the dim-statement.

```
dim-statement = "DIM" array-declaration (","
     array-declaration)*
array-declaration = array-name "(" dimension-bounds
     ("," dimension-bounds)* ")"
array-name = identifier
dimension-bounds = (numeric-expression ":")?
     numeric-expression
```

If the dimension-bounds for some dimension of an array consists of a single numeric-expression, the value of that expression is the upper bound of the dimension; the lower bound is that specified in the option-base-statement of the containing external-routine, or zero if no option-base-statement is present. Thus, in the absence of an option-base-statement, "DIM A(10)" makes A an 11-element array whose subscripts are zero through ten.

The dim-statement is an executable statement; it changes the size, but not the rank, of an array. (The <u>rank</u> of an array is the number of its dimensions.) Before any dim-statement has been executed, the lower bound of each dimension of every array is that specified in the option-base-statement of the containing external-routine, or zero if no option-base-statement is present; the upper bound of each dimension of every array is 10. Thus, depending on option base, each n-dimensional array is initially of size 10^n (base 1) or 11^n (base 0), where n is the number of dimensions. The lower and upper bounds of each dimension are initially 0 and 10, respectively.

Execution of a dim-statement preserves values wherever subscripts are preserved. That is, if A is an n-dimensional array and S is a list of n numeric-expressions separated by commas, and if A(S) is a legal reference to the array both before and after the execution of a dim-statement, then execution of the dim-statement will not change the value of A(S).

If a formal-array is redimensioned by a called routine, the dimension bounds of the corresponding actual array are changed in the calling routine. The redimensionings of both the actual- and the formal-arrays are effective immediately.

3.5.2.2.5.3 Erase-Statement

The erase-statement frees storage occupied by arrays.

```
    erase-statement = "ERASE" array-name (
         "," array-name)*
```

The array-names are the names of the arrays whose storage is to be freed. After the erase-statement has executed, both the lower and upper bounds of each dimension are zero or one, depending on option base. The value of the single element of each erased array is set to zero or null according as the array is a numeric or string-array.

If a formal-array is erased, so is the actual-array which was passed to it; the changes are immediately effective.

3.5.2.2.5.4 Randomize-Statement

The randomize-statement reseeds the random number generator.

```
    randomize-statement = "RANDOMIZE" numeric-expression?
```

If the numeric-expression is provided, it is used to generate the seed. Sequences of pseudo-random numbers may be reproduced by reseeding with an invariant numeric-expression. If the numeric-expression is omitted, the standard input source is queried for a seed. A randomize-statement without a numeric-expression is equivalent to

```
INPUT "Random Number seed?  ",SEED : RANDOMIZE SEED
```

3.5.3 BLOCKS

Virtual BASIC includes several multi-statement structures. This section describes those structures.

A block is a sequence of block-elements separated by statement-bounderies.

```
block = block-element
     (statement-boundary block-element)*
```

A block-element is either an unstructured-statement or a complete structure like a FOR loop or an internal-function.

```
block-element = unstructured-statement /
     line-if-statement /
     for-block /
     if-block /
     internal-function /
     internal-subroutine /
     while-block
unstructured-statement = assignment-statement /
     control-statement /
     declarative-statement /
     io-statement /
     data-initialization-statement /
     miscellaneous-executable-statement
```

3.5.3.1 Line-If-Statement

A line-if-statement is an IF whose effects are confined to a single source line.

```
line-if-statement = "IF" numeric-expression ("THEN"
     (line-block / line-number) / "GOTO" line-number)
     ("ELSE" (line-block / line-number))?
```

The consequent of the IF begins either with "THEN" or with "GOTO". If the consequent begins with "GOTO", it is a single unconditional jump to the indicated line-number. If the consequent begins with "THEN", the "THEN" is followed either by a line-number, in which case it is a single unconditional jump to the indicated line-number, or else by a line-block, which is a non-empty sequence of block-elements separated by colons.

```
    line-block = block-element (":" block-element)*
```

The alternate of a line-if-statement begins with "ELSE"; what follows the else has the same syntax as what follows the "THEN" variant of the consequent.

A line-if-statement is executed as follows. The numeric-expression is evaluted. If it is non-zero, the consequent of the if is executed. Otherwise, the alternate is executed.

It is a consequence of the syntax that if any part of a block appears in either the consequent or the alternate of a line-if-statement, then all of it must appear there. In other words, line-if-statements must be well nested with the other, potentially multi-line, structures of Virtual BASIC.

3.5.3.2 For- and Next-Statements

A for-block consists of a for-statement, the corresponding next-statement, and all the code in between.

```
    for-block = for-statement
         delimited-optional-blk next-statement
    delimited-optional-blk = statement-boundary
         (block statement-boundary)?
```

A for-statement specifies the control-variable, its initial value, the limit value, and, either explicitly or by default, the step.

```
    for-statement = "FOR" control-variable "="
         numeric-expression "TO" numeric-expression
         ("STEP" numeric-expression)?
    control-variable = numeric-scalar
```

The first numeric-expression is the initial value, the second is the limit value, and the third, if present, is the step.

The next-statement closes one or more FOR loops.

```
    next-statement = "NEXT" control-variable-list?
    control-variable-list = control-variable
         ("," control-variable)?
```

A next-statement which specifies no control-variable closes the most recent unclosed FOR loop. If control-variables are specified on a next-statement, they must be the control-variables of the most recent unclosed FOR loop, the next most recent, and so on, in that order. All the FOR loops whose control-variables are mentioned on a next-statement are closed by that next-statement.

The statements between a for-statement and its next-statement are executed repeatedly. After each execution, the control-variable of the loop is increased by the step value. Barring other alteration of the value of the control-variable, the loop is executed

    max((l + s - i) \ s, 0)

times, where l is the limit value, s is the step, and i is the initial value. In any case, execution of the loop ends after the first pass which leaves

    (control-variable - l)*SGN(s) > 0

where SGN returns -1, 0, or 1 according as its argument is less than, equal to, or greater than zero.

Note that the expressions for limit value and step are evaluated once on entry to the loop; alteration within the loop of variables used in these expressions has no effect on the number of executions of the loop.

3.5.3.3 Block-If, Elseif-, Else-, and Endif-Statements

An if-block consists of the keyword IF followed by an if-body. An if-body is a numeric-expression, the keyword THEN, a consequent, and an if-tail in that order.

    if-block = "IF" if-body
    if-body = numeric-expression "THEN" consequent if-tail

An if-tail is either the keyword ELSEIF followed by an if-body or else an optional ELSE and alternate followed unconditionally by an ENDIF.

    if-tail = "ELSEIF" if-body / ("ELSE" alternate)? "ENDIF"

Consequents and alternates are just blocks. They may be empty; they include their delimiting statement-boundaries.

```
consequent = delimited-optional-blk
alternate = delimited-optional-blk
```

Execution of an if-body proceeds as follows. The numeric-expression is evaluated. If it is non-zero, the consequent is executed and control passes to the statement following the if-body; if it is zero, the if-tail is executed according to the rule given in the following paragraph.

Execution of an if-tail proceeds as follows. If the if-tail begins with an ELSEIF, the if-body following the ELSEIF is executed according to the rule given in the preceding paragraph. Otherwise, the alternate, if any, is executed and control passes to the statement following the if-tail.

3.5.3.4 While- and Wend-Statements

A while-block begins with the keyword WHILE and ends with the keyword WEND. In between are a numeric-expression and a while-body in that order.

```
while-block = "WHILE" numeric-expression while-body "WEND"
```

A while-body is a (possibly empty) block together with its delimiting statement boundaries.

```
while-body = delimited-optional-blk
```

Execution of a while-block proceeds as follows. The numeric-expression is evaluated. If it is non-zero, the while-body is executed and the while-block is re-executed. Otherwise, control passes to the statement following the while-block.

3.5.4 ROUTINES

In this document, the word "routine" is a generic term which encompasses internal and external functions and subroutines. It does not include expression functions.

Routines are either internal or external. Internal routines have access to all their host's data and, excepting only certain formal parameters, have no local data of their own. Internal routines may not contain embedded internal routines.

Functions may have side effects. Recursive calls are permitted. Entire arrays may be passed as actual parameters.

A routine may have at most 255 parameters. Routine parameters are passed by address in Virtual BASIC. Corresponding actual and formal parameters of user routines must be of like type. In particular, it is an error to pass an integer actual parameter to a user routine's real formal parameter or to pass a real actual to an integer formal.

Any formal parameter may be modified by its declaring routine. If the corresponding actual parameter is an array or a scalar, the modification is effective in the calling routine; otherwise, it is not. The presence of parentheses will not protect an actual parameter from modification by a called routine. X is equally liable to modification by SUB, whether the call is

```
CALL SUB(X)
```

or

```
CALL SUB((X))
```

If an actual parameter is a non-trivial expression, an array element, a substring, or constant, its value in the calling routine will be unaffected by any modifications which the called routine may make to the corresponding formal parameter. Thus, returning to the immediately preceding example,

```
CALL SUB(X+0.0)
```

would have protected the value of X in the calling routine.

When parameter passing creates aliases, the order of modifications is preserved. Thus,

```
SUB SUB(X,Y)
PRINT X,Y,A
X = 2 : PRINT X,Y,A
Y = 3 : PRINT X,Y,A
A = 4 : PRINT X,Y,A
END SUB
'
A = 1
CALL SUB(A,A)
```

would result in

```
1 1 1
```

```
2 2 2
3 3 3
4 4 4
```

on the standard output file.

If the dimension-bounds of a formal-array are altered in a called routine, the dimension-bounds of the corresponding actual-array are altered in the calling routine. In other words, clear-, dim-, and erase-statements executed by the called routine are effective in the calling routine.

To summarize, routine parameters are passed by address. Corresponding actual and formal parameters must be of like type. Actuals which are arrays or scalars are subject to modification (including, in the case of arrays, modification of dimension bounds) by the called routine. Actuals which are constants, array elements, substrings, or non-trivial expressions are not. Parameter passing can create aliases; however, the user can count on the order of references remaining as specified in the source code.

3.5.4.1 External-Routines

A Virtual BASIC source-deck consists of a sequence of external-routines separated by statement-boundaries. The sequence is optionally preceded by a line-number.

```
source-deck = line-number?  external-routine
     (statement-boundary external-routine)*
```

A line-number is an integer. Leading zeros in line-numbers are insignificant.

```
line-number = integer
```

External-routines are contained in no other routine. They share data with each other by means of parameters and COMMON only. External-routines are separately compilable. The semantics of external-routines are unaffected by other external-routines which may be part of the same source-deck. In particular, compilation defaults are reinitialized for each external-routine. Thus, for example, the OPTION BASE is initially 0 in an external-subroutine, even though it may have been 1 in the immediately preceding external-routine.

```
external-routine = main-program /
     external-function / external-subroutine
```

A main-program is a block.

```
    main-program = block
```

If the last statement of a main-program is executed and does not cause a transfer of control, then execution of the program terminates.

An external-function consists of an external-function-statement, a non-empty body, and an end-function-statement.

```
    external-function = external-function-statement
         delimited-block end-function-statement
    delimited-block = statement-boundary block
         statement-boundary
```

(The body of an external-function is non-empty because, at the minimum, an assignment to the function name is required.)

An external-subroutine consists of an external-sub-statement, a possibly empty body, and an end-sub-statement.

```
    external-subroutine = external-sub-statement
         delimited-optional-blk end-sub-statement
```

There is no program-statement in Virtual BASIC. External-functions begin with an external-function-statement and end with an end-function-statement. External-subroutines begin with an external-sub-statement and end with an end-sub-statement. If an end-function- or end-sub-statement is executed, it behaves as an exit-function- or exit-sub-statement respectively.

```
    external-function-statement = "EXTERNAL" "FUNCTION"
         function-name formal-parameter-list?
    external-sub-statement = "EXTERNAL" "SUB"
         subroutine-name formal-parameter-list?
    formal-parameter-list = "(" formal-parameter
         ("," formal-parameter)* ")"
    end-function-statement = "END" "FUNCTION"
    end-sub-statement = "END" "SUB"
```

A formal-parameter is either a formal-scalar or a formal-array. The number of dimensions of a formal-array is one more than the number of commas between the parentheses. The dimension-bounds of a formal-array are those of the corresponding actual-array.

```
    formal-parameter = formal-scalar / formal-array
```

```
formal-scalar = identifier
formal-array = identifier "(" ","* ")"
```

Subroutine-names must be plain, unadorned names. Function-names may be pre-typed names.

```
subroutine-name = name
function-name = string-function-name /
     numeric-function-name
string-function-name = string-identifier
numeric-function-name = numeric-identifier
```

3.5.4.2 Internal Routines

Internal-routines are embedded within external-routines. An internal routine may not contain another internal routine. Internal-routines have access to all the data of the external-routines in which they are embedded. They have no local data of their own. Only formal parameters to which a non-trivial expression, array element, substring, or constant was passed are truly local to an internal routine.

The definitions of internal-functions and internal-subroutines look like those of external-functions and -subroutines except that the "EXTERNAL" is omitted from the routines' headers and internal routines are prohibited in their bodies.

```
internal-function = internal-function-statement
     statement-boundary internal-block statement-boundary
     end-function-statement
internal-subroutine = internal-sub-statement
     statement-boundary (internal-block
     statement-boundary)?
     end-sub-statement
internal-function-statement = "FUNCTION" function-name
     formal-parameter-list?
internal-sub-statement = "SUB" subroutine-name
     formal-parameter-list?
internal-block = internal-block-element
     (statement-boundary internal-block-element)*
internal-block-element = unstructured-statement /
     line-if-statement /
     for-block /
     if-block /
     while-block
```

Declarative statements in effect in the host external-routine remain in effect in an internal routine; declarative statement which are contained within an internal-routine affect not just the internal-routine but the entire containing external-routine.

If control reaches the first statement of an internal routine when the routine has not been invoked by name, it passes directly to the statement following the last of the internal routine. The statements in the internal routine are not executed. If the last line of the internal routine is the last line of a main-program, execution of the program terminates.

3.6 LIBRARY

This section describes the functions available in the Virtual BASIC runtime library. Assignments of the values of the functions to program variables are shown to illustrate function references. These examples are not intended to be part of the definition of Virtual BASIC syntax.

3.6.1 MATHEMATICAL FUNCTIONS

3.6.1.1 ABS

The ABS function returns the absolute value of its argument.

```
numeric-variable = ABS( numeric-expression )
```

Any numeric-expression may be used as the argument of the ABS function. An exception is raised if the argument is machine indefinite or infinite. The type of the value returned is the type of the argument.

3.6.1.2 ACOS

The ACOS function returns a real number the value of which is the inverse circular cosine of its argument.

```
numeric-variable = ACOS( numeric-expression )
```

The argument of ACOS may be any numeric-expression, but the function is always evaluated as real. If the absolute value of the argument of ACOS is not less than or equal to 1 or is indefinite, an exception is raised. The argument is presumed to be in radians.

3.6.1.3 ASIN

The ASIN function returns a real number the value of which is the inverse circular sine of its argument.

```
numeric-variable = ASIN( numeric-expression )
```

The argument of ASIN may be any numeric-expression, but the function is always evaluated as real. If the absolute value of the argument is not less than or equal to 1 or is indefinite, an exception is raised. The argument is presumed to be in radians.

3.6.1.4 ATN

The ATN function returns a real number the value of which is the inverse circular tangent of its argument.

```
numeric-variable = ATN( numeric-expression )
```

The argument of ATN may be any numeric-expression, but the function is always evaluated as real. If the argument is indefinite, an exception is raised. The argument is presumed to be in radians.

3.6.1.5 CDBL

The CDBL function returns a real representation of the value of its argument.

```
numeric-variable = CDBL( numeric-expression )
```

The CDBL function is included in Virtual BASIC only for compatibility. This function is identical to the CSNG function (see the section of this document discussing data types).

3.6.1.6 CEIL

The CEIL function returns an integer the value of which is the smallest integer not less than the value of its argument.

```
numeric-variable = CEIL( numeric-expression )
```

The argument of CEIL may be any numeric-expression. An exception is raised if the smallest integer not less than its argument cannot be represented by a NOS/VE integer or if the value of the argument is machine indefinite or infinite.

3.6.1.7 CINT

The CINT function returns an integer the value of which is that of its argument rounded to the nearest integer.

```
numeric-variable = CINT( numeric-expression )
```

The argument of CINT may be any numeric-expression. An exception is raised if the value of the argument is indefinite or outside the range representable by a NOS/VE integer.

3.6.1.8 COS

The COS function returns a real number the value of which is the circular cosine of its argument.

```
numeric-variable = COS( numeric-expression )
```

The argument of COS may be any numeric-expression, but the function is always evaluated as real. An exception is raised if the value of the argument is machine indefinite, infinite, or has an absolute value greater than or equal to 2^47. The argument is presumed to be in radians.

3.6.1.9 COSH

The COSH function returns a real number the value of which is the hyperbolic cosine of its argument.

```
numeric-variable = COSH( numeric-expression )
```

The argument of COSH may be any numeric-expression, but the function is always evaluated as real. An exception is raised if the value of the argument is machine indefinite, infinite, or has an absolute value greater than or equal to 4095 * LOG(2).

3.6.1.10 CSNG

The CSNG function returns the real representation of the value of its argument.

```
numeric-variable = CSNG( numeric-expression )
```

The argument of CSNG may be any numeric-expression. If the value of the argument is integral, the function converts it to a real value. If the value of the argument is real, the function returns that value. An exception is raised if the value of the argument is machine indefinite or infinite.

3.6.1.11 DEG

The DEG function returns a real number the value of which is the number of degrees in its radian argument.

```
numeric-variable = DEG( numeric-expression )
```

The argument of DEG may be any numeric-expression, but the function is always evaluated as real. An exception is raised if the value of the argument is machine indefinite or if the absolute value of the argument multiplied by (180/PI) cannot be represented by a NOS/VE floating-point number.

3.6.1.12 EXP

The EXP function returns a real number the value of which is the exponential of its argument.

```
numeric-variable = EXP( numeric-expression )
```

The argument of EXP may be any numeric-expression, but the function is always evaluated as real. An exception is raised if the value of the argument is machine indefinite or if the absolute value of the argument is greater than or equal to 4095 * LOG(2).

3.6.1.13 FP

The FP function returns a real number the value of which is the fractional part of its argument.

    numeric-variable = FP(numeric-expression)

The argument of FP may be any numeric-expression, but the function is always evaluated as real. If the value of the argument is integral, the function returns zero. The fractional part of a real argument with an absolute value greater than 10^18 is always zero. An exception is raised if the value of the argument is machine indefinite or infinite.

3.6.1.14 FIX

The FIX function returns an integer the value of which is its argument truncated to an integer.

    numeric-variable = FIX(numeric-expression)

The argument of FIX may be any numeric-expression. If the expression is integral, the function returns the value of the expression. If the expression is real, the function returns the value of the expression truncated to an integer. An exception is raised if the value of the argument is machine indefinite or if the absolute value of the argument is outside the range representable by NOS/VE integers.

3.6.1.15 INT

The INT function returns an integer the value of which is the largest integer not greater than the value of its argument.

    numeric-variable = INT(numeric-expression)

The argument of INT may be any numeric-expression. If the expression is integral, the function returns the value of the expression. If the expression is real, the function returns the largest integer that is not greater than the value of the argument. An exception is raised if the value of the argument is machine indefinite or if the absolute value of the argument is outside the range representable by NOS/VE integers.

3.6.1.16 LOG

The LOG function returns a real number the value of which is the natural logarithm of its argument.

```
    numeric-variable = LOG( numeric-expression )
```

The argument of LOG may be any numeric-expression, but the function is always evaluated as real. An exception is raised if the value of the argument is machine indefinite, infinite, or not greater than zero.

3.6.1.17 LOG10

The LOG10 function returns a real number the value of which is the common logarithm of its argument.

```
    numeric-variable = LOG10( numeric-expression )
```

The argument of LOG10 may be any numeric-expression, but the function is always evaluated as real. An exception is raised if the value of the argument is machine indefinite, infinite, or not greater than zero.

3.6.1.18 MAX

The MAX function returns the algebraically larger of the values of its arguments.

```
    numeric-variable = MAX( numeric-expression,
                            numeric-expression )
```

The arguments of MAX may be any numeric-expressions. An exception is raised if the value of an argument is machine indefinite or infinite. The type of the value returned is real if either operand is real, integer if both operands are of type integer.

3.6.1.19 MIN

The MIN function returns the algebraically smaller of the values of its arguments.

```
numeric-variable = MIN( numeric-expression,
                        numeric-expression )
```

The arguments of MIN may be any numeric-expressions. An exception is raised if the value of an argument is machine indefinite or infinite. The type of the value returned is real if either operand is real, integer if both operands are of type integer.

3.6.1.20 RAD

The RAD function returns a real number the value of which is the number of radians in the value of its argument. The argument is presumed to be in degrees.

```
numeric-variable = RAD( numeric-expression )
```

The argument of RAD may be any numeric-expression, but the function is always evaluated as real. An exception is raised if the value of the argument is machine indefinite or if the absolute value of the argument multiplied by (PI/180) cannot be represented by a NOS/VE floating-point number.

3.6.1.21 RND

The RND function returns a random real number the value of which is between zero and 1.0.

```
    numeric-variable = RND( numeric-expression )
or
    numeric-variable = RND
```

The RND function may be called with or without an argument. If specified, the argument may be any numeric-expression, but the function is evaluated as real.

If RND is called with a negative argument, the random number generator is reseeded. (The random number generator also may be reseeded by the randomize-statement.) A given negative argument always induces the same pseudo-random sequence. To induce variation in the randomness of the sequence, a mutable seed (such as -VAL(RIGHT$(TIME$,2))) should be specified.

If RND is called with no argument or an argument greater than zero, the next value of the pseudo-random sequence is returned. If an argument equal to zero is specified, the most recently returned result of the function is repeated.

3.6.1.22 SGN

The SGN function returns the sign of its argument.

```
numeric-variable = SGN( numeric-expression )
```

The argument of SGN may be any numeric-expression; the value returned is always an integer. If the value of the argument is greater than zero, the value 1 is returned. If the value of the argument is equal to zero, the value 0 is returned. If the value of the argument is less than zero, the value -1 is returned. An exception is raised if the value of the argument is machine indefinite or infinite.

3.6.1.23 SIN

The sin function returns a real number the value of which is the circular sine of its argument.

```
numeric-variable = SIN( numeric-expression )
```

The argument of the SIN function may be any numeric-expression, but the function is always evaluated as real. An exception is raised if the argument is machine indefinite or if the absolute value of the argument is greater than or equal to 2^47. The argument is presumed to be in radians.

3.6.1.24 SINH

The SINH function returns a real number the value of which is the hyperbolic sine of its argument.

```
numeric-variable = SINH( numeric-expression )
```

The argument of SINH may be any numeric-expression, but the function is always evaluated as real. An exception is raised if the value of the argument is machine indefinite or if the absolute value of the argument is greater then or equal to 4095 * LOG(2).

3.6.1.25 SQR

The SQR function returns a real number the value of which is the square root of its argument.

```
numeric-variable = SQR( numeric-expression )
```

The argument of SQR may be any numeric-expression, but the function is always evaluated as real. An exception is raised if the value of the argument is machine indefinite, less than zero, or infinite.

3.6.1.26 TAN

The TAN function returns a real number the value of which is the circular tangent of its argument.

```
numeric-variable = TAN( numeric-expression )
```

The argument of TAN may be any numeric-expression, but the function is always evaluated as real. An exception is raised if the value of the argument is machine indefinite or if the absolute value of the argument is greater than or equal to 2^47. The argument is presumed to be in radians.

3.6.1.27 TANH

The TANH function returns a real number the value of which is the hyperbolic tangent of its argument.

```
numeric-variable = TANH( numeric-expression )
```

The argument of TANH may be any numeric-expression, but the function is always evaluated as real. An exception is raised if the value of the argument is machine indefinite.

3.6.2 STRING AND MISCELLANEOUS FUNCTIONS

3.6.2.1 ASC

The ASC function returns an integer the value of which is the ASCII ordinal of the first character of its argument.

```
numeric-variable = ASC( string-expression )
```

The argument of ASC may be any string-expression. The value returned is an integer in the range zero through 255. An exception is raised if the value of the argument is null.

3.6.2.2 CHR$

The CHR$ function returns the literal character whose ASCII ordinal is the value of the argument.

```
string-variable = CHR$( numeric-expression )
```

The argument of CHR$ may be any numeric-expression, but the value used is rounded to the nearest integer. An exception is raised if the rounded value of the argument is not in the range zero through 255. The value returned by the function is a single-character string.

3.6.2.3 ERL

The ERL function returns the integer value associated with the line whose execution raised the most recent exception. If no exception has been raised in the current invocation of the routine, ERL returns zero.

Note that if an exception is raised by a statement which precedes the first numbered line in an external-routine, ERL will return zero.

3.6.2.4 ERR

The ERR function returns the ordinal of the current exception in an external-routine. Upon entry to any routine, the value of ERR is zero. ERR is set to an integer code denoting the particular error condition when an exception is raised. Its non-zero value persists until it is cleared by the execution of a resume-statement. An attempt by the user to store to ERR yields a compilation error.

ERR, together with ERL, frequently is useful in user-specified exception handling code. (See the on-error-goto-statement.)

3.6.2.5 HEX$

The HEX$ function returns the printable representation of the hexadecimal value of its argument.

```
string-variable = HEX$( numeric-expression )
```

The argument of HEX$ may be any numeric-expression, but the value used is rounded to the nearest integer. An exception is raised if the rounded value cannot be represented as a NOS/VE integer. Negative values are shown in two's complement representation. The length of the string returned by the function is the shortest representation that allows the most significant digit of a positive value to be less than eight or the most significant digit of a negative value to be greater than seven.

3.6.2.6 INSTR

The INSTR function searches for the occurrence of one string within another and returns the position at which the first match is found.

```
numeric-variable = INSTR( numeric-expression,
                          string-expression,
                          string-expression )
```

or

```
numeric-variable = INSTR( string-expression,
                          string-expression )
```

The optional numeric-expression denotes the beginning character position for the search. If none is specified, the first string is searched from its beginning. Any numeric-expression may be specified, but the value used is rounded to the nearest integer. If a specified value rounds to less than 1 or greater than the maximum string length, an exception is raised.

If a beginning position is specified that is beyond the length of the value of the first string-expression, the value of the first string-expression is null, or the value of the second string-expression is not found, the function returns zero. If the value of the second string-expression is null, it is "found" at the first position searched, and the function returns the ordinal of that position. If the value of the second string-expression is found within the value of the first string-expression, its beginning character position is returned by the function.

3.6.2.7 LBOUND

The LBOUND function returns the minimum value of an array's subscript.

```
    numeric-variable = LBOUND ( array-name,
                                numeric-expression )
or
    numeric-variable = LBOUND ( array-name )
```

The numeric-expression parameter may be any numeric-expression, but the value used is rounded to the nearest integer. An exception is raised if the rounded value of the numeric-expression cannot be represented as a NOS/VE integer, if it is less than one, or if it is greater than the number of dimensions of the array denoted by array-name.

The function returns an integer whose value is the minimum value of the array's n-th subscript, where n is the rounded value of the numeric-expression. If the array has only one dimension, the numeric-expression parameter may be omitted; a value of one is inferred.

3.6.2.8 LCASE$

The LCASE$ function returns a string equal in length and value to its argument except that each uppercase alphabetic character is replaced by its lowercase homologue.

```
    string-variable = LCASE$( string-expression )
```

The argument of LCASE$ may be any string-expression. Characters of the value of the argument that are not uppercase alphabetic are not substituted in the returned value of the function.

3.6.2.9 LEFT$

The LEFT$ function returns a string whose value is the left-most n characters of the value of the string-expression passed it, where n is the value of the numeric-expression passed it.

```
    string-variable = LEFT$( string-expression,
                             numeric-expression )
```

The first argument of LEFT$ may be any string-expression. The second argument may be any numeric-expression, but the value used is rounded to the nearest integer. If the value of the numeric-expression is negative or cannot be rounded to a NOS/VE integer, an exception is raised. If the value of the numeric-expression is greater than the length of the value of the string-expression, the entire value of the string-expression is returned. If the value of the numeric-expression is zero, a null string is returned.

3.6.2.10 LEN

The LEN function returns the length in characters of its argument.

```
    numeric-variable = LEN( string-expression )
```

The argument of LEN may be any string-expression. The value returned is a non-negative integer.

3.6.2.11 MID$

The MID$ function returns a substring of its first argument as specified by its numeric arguments.

```
    string-variable = MID$( string-expression,
                            numeric-expression )
or
    string-variable = MID$( string-expression,
                            numeric-expression,
                            numeric-expression )
```

Note that MID$ is a peculiar function in that it may appear on either side of an equal sign. In either case, it is used to indicate a kind of substring addressing. When MID$ appears on the right-hand side of an assignment, it behaves as a function returning a string value. When it appears on the left-hand side of an assignment, it indicates that the assignment is to be to a substring of its argument, which must be either a string-expression or a string-array-element.

The first argument of the MID$ function may be any string-expression. The numeric parameters of the function may be any numeric-expressions; the values used are rounded to the nearest integers. If the rounded value of either numeric parameter cannot be represented by a NOS/VE integer, an exception is raised. If the value of the first numeric-expression is less than 1, or if the value of the second numeric-expression is less than zero, an exception is raised.

The first numeric-expression indicates the beginning character position of the substring to be addressed in the value of the string-expression argument. If the value of this numeric-expression is greater than the length of the value of the string-expression, the MID$ function uses a null string.

The second numeric-expression indicates the length of the substring to be addressed. If this parameter is omitted, or if it is greater than the number of characters from the specified beginning position through the end of the source string, then all the characters from the beginning position through the end of the source string are addressed. If this parameter is specified, but rounds to zero, a null string is addressed.

3.6.2.12 OCT$

The OCT$ function returns the printable representation of the octal value of its argument.

```
string-variable = OCT$( numeric-expression )
```

The argument of OCT$ may be any numeric-expression, but the value used for the function is rounded to the nearest integer. An exception is raised if the rounded value cannot be represented as a NOS/VE integer. Negative values are shown in two's complement representation. The length of the string returned by the function is the shortest representation that allows the most significant digit of a positive value to be less than four or the most significant digit of a negative value to be greater than three.

3.6.2.13 RIGHT$

The RIGHT$ function returns a string whose value is the right-most n characters of the value of its string-expression argument, where n is the value of its numeric-expression argument.

```
string-variable = RIGHT$( string-expression,
                          numeric-expression )
```

The string-expression argument of RIGHT$ may be any string-expression. The numeric-expression argument may be any numeric-expression, but the value used is rounded to the nearest integer. An exception is raised if the value of the numeric-expression is negative or if it cannot be represented as a NOS/VE integer.

If the value of the numeric-expression is greater than or equal to the length of the value of the string-expression, the function returns the value of the string-expression. If the value of the numeric-expression is zero, the function returns a null string.

3.6.2.14 SPACE$

The SPACE$ function returns a string of blank characters.

```
string-variable = SPACE$( numeric-expression )
```

The argument of SPACE$ may be any numeric-expression, but the value used is rounded to the nearest integer. An exception is raised if the rounded value is negative or greater than the maximum string length. The length of the string returned by the function is the rounded value of the numeric-expression.

3.6.2.15 STR$

The STR$ function returns a string the value of which is a printable representation of the value of its argument.

```
string-variable = STR$( numeric-expression )
```

The argument of STR$ may be any numeric-expression. An exception is raised if the value of the numeric-expression is machine indefinite or infinite. The value of the string returned by the function is the same representation of the numeric value that would result from the specification of the numeric-expression in a print-list of a print-statement.

3.6.2.16 STRING$

The STRING$ function returns a uniform string of characters.

```
    string-variable = STRING$( numeric-expression,
                               numeric-expression )
or
    string-variable = STRING$( numeric-expression,
                               string-expression )
```

A numeric-expression passed to the STRING$ function may be any numeric-expression, but the value used is rounded to the nearest integer. If the value of any numeric-expression is negative or cannot be represented as a NOS/VE integer, an exception is raised. If the value of the first numeric-expression is greater than the maximum string length, an exception is raised. If the function is called with a second numeric-expression parameter and the value of the numeric-expression is greater than 255, an exception is raised. The string-expression parameter may be any string-expression. If the value of the string-expression is null, an exception is raised.

The value of the first numeric-expression parameter determines the length of the value returned. If the second parameter is a numeric-expression, the string returned by STRING$ is filled with the ASCII character whose ordinal is the value of the numeric-expression. If the second parameter is a string-expression, the string returned is filled with the first character of the value of the string-expression.

3.6.2.17 UBOUND

The UBOUND function returns the maximum value of an array's subscript.

```
    numeric-variable = UBOUND ( array-name,
                                numeric-expression )
or
    numeric-variable = UBOUND ( array-name )
```

The numeric-expression parameter may be any numeric-expression, but the value used is rounded to the nearest integer. An exception is raised if the rounded value of the numeric-expression cannot be represented as a NOS/VE integer, if it is less than one, or if it is greater than the number of dimensions of the array denoted by the array-name.

The function returns an integer whose value is the minimum value of the array's n-th subscript, where n is the rounded value of the numeric-expression. If the array has only one dimension, the numeric-expression may be omitted; a value of one is assumed.

3.6.2.18 UCASE$

The UCASE$ function returns a string equal in length and value to its argument except that each lowercase alphabetic character is replaced by its uppercase homologue.

```
string-variable = UCASE$( string-expression )
```

The argument of UCASE$ may be any string-expression. Characters of the value of the string-expression that are not lowercase alphabetic are not substituted in the returned value of the function.

3.6.2.19 VAL

The VAL function returns the numeric value of a string-expression.

```
numeric-variable = VAL( string-expression )
```

The argument of VAL may be any string-expression. The function returns the value of the first nonblank character sequence of the value of the string which can be interpreted as a number. If the value of the string-expression is null, or if no leading substring of the value after any leading blanks can be interpreted as a number, the function returns the value zero.

3.7 IDENTIFIER DECLARATION

Identifiers in Virtual BASIC programs frequently are declared by default. A single identifier refers to a single object, !

except that arrays and scalars of the same name may coexist. !

The general rule about declarations in Virtual BASIC is this: an object must be fully declared at or before its first use. More precisely, an object must be fully declared in the substring of the source text which begins with the beginning of the containing external-routine and which ends with, and includes, the first use. Inclusion of the first use in the declaring substring is important since most declaration in Virtual BASIC is by default. For example,

```
A(5) = 5
```

sensibly may be the first line of a program. If it is, it declares, by default, that A is a one-dimensional real array of size 11 whose elements are subscripted 0 through 10.

There is no requirement that declarative statements be grouped together or that they precede executable statements. The compiler will gather declarative information as it consumes source text and update the declaration status of identifiers as it does so. Table 3.7B below describes the effects of various appearances of an identifier on the declaration status of the identifier.

The declaration status of the identifier is represented by an eight-element vector. The first seven elements of the vector contain values which are denoted by "+", "?", "-", "+?", and "-?". These values mean, roughly, "is", "might be", "can't be", "is unless known not to be", and "isn't unless known to be." The eighth element of the vector contains either "?", meaning "don't know", or a thing called a parameter list characterization, optionally followed by a question mark. A parameter list characterization can be thought of as a character string constructed according to the following little grammar:

```
parameter-list-characterization = "(" parameter-list?  ")"
parameter-list = parameter ("," parameter)*
parameter = expression / array
expression = "E" ("%" / "!"  / "$")?
array = "A" ("%" / "!"  / "$") "(" ","* ")"
```

A parameter list characterization is a straightforward representation of what the legal parameters (or subscripts) of an identifier are. "E" represents an expression; "A" represents an array. If an "E" or an "A" is followed by a percent sign, then the expression or array is of type integer; if by an exclamation point, then of type real; if by a dollar sign, then of type string. If an "E" is followed by none of these characters, then either an integer expression or a real expression is acceptable. The number of dimensions of an array parameter is one more than the number of commas between the parentheses.

A parameter list characterization followed by a question mark means, roughly, "parameters are as shown unless they are known to be otherwise."

The eight elements of the vector represent attributes of the identifier. The attributes of the identifier are, in order, "is a scalar", "is an array", "is a function", "is an expression-function", "is a subroutine", "non-array referent is internal", "array referent is internal", and "parameters are as shown." The initial value of the vector, for any identifier, is "? ? ? ? ? ? ? ?"; this says, as you might expect, that nothing is known about the identifier.

For a given appearance of an identifier A, Table 3.7B gives an eight-element vector. The declaration status vector of A and the vector from Table 3.7B are combined, element by element, to yield an updated declaration status vector. The combination operator is defined in Table 3.7A.

Table 3.7A:

+ + +	+ ? +	+ - X	+ P .	
? + +	? ? ?	? - -	? P P	
- + X	- ? -	- - -	- P .	
+? + +	+? ? +	+? - -	+? P .	
-? + +	-? ? -	-? - -	-? P .	
P + .	P ? P	P - .	P P P	P Q X
P? + .	P? ? P	P? - .	P? P P	P? Q Q

The first value in each triplet is the value from the table; it reflects what has just been learned about the identifier. The second is the value from the declaration status vector for the identifier; it reflects what has been learned about the identifier heretofore. The third represents what can be inferred from what was known heretofore and what was just learned. An X as a third element indicates some kind of error. Thus, for example, the entry "+ - X" (middle of first line) says that if an appearance of an identifier implies that it has a some attribute, and the identifier is already known not to have that attribute, there is some sort of error in the program. A dot as the third element of a triplet means that there is no way that a situation can arise which requires combination of the first and second elements of the triplet. P and Q represent parameter list characterizations; they are assumed to be distinct.

3.0 FEATURE DESCRIPTION
3.7 IDENTIFIER DECLARATION

Here, finally, is Table 3.7B.

Table 3.7B:

Appearance of A	Status Vector Update for A							
	SCALAR	ARRAY	FN	EXP FN	SUB	INT	INT ARR	PLC
COMMON A	+	?	-	-	-	-	?	?
COMMON A()	?	+	-	-	-	?	-	?
DECLARE FUNCTION A	-	-	+	-	-	+	?	?
DECLARE EXTERNAL ...								
... FUNCTION A	-	-	+	-	-	-	?	?
FUNCTION A(X!(),Y$())	-	-	+	-	-	+	?	(A!(),A$())
DEF A(X%,S$)=X%+LEN(S$)	-	-	+	+	-	+	?	(E,E$)
DIM A(4,4)	?	+	-	-	-	?	?	(E,E)
LET A = 5	+	?	-	-	-	+?	?	?
LET A(1,2,3) = 0	?	+	-	-	-	?	+?	(E,E,E)
LET X = A(1,"")	-	-	+	-?	-	+?	?	(E%,E$)?
CALL A	-	-	-	-	+	+?	?	()
CALL A(X,Y!(,,))	-	-	-	-	+	+?	?	(E!,A!(,,))
CALL X(A)	+?	?	-?	-?	-	+?	?	?
CALL X(A(5,3))	?	+?	-?	-?	-	+?	?	(E,E)?
CALL X(A(5,"3"))	-	-	+	-?	-	+?	?	(E%,E$)?
READ A	+	?	-	-	-	+?	?	?
PRINT A	+?	?	-?	-?	-	+?	?	?
READ A(1)	?	+	-	-	-	?	+?	(E)
PRINT A(1)	?	+?	-?	-?	-	?	+?	(E)?

As an example of the use of Tables 3.7A and 3.7B, consider their application to the following program.

```
COMMON A()
LET A(3,5) = 0
LET A = A(3,5)
CALL A
```

Here is the program again, this time including the declaration status of the identifier A before and after each statement.

```
REM             ? ? ? ? ? ? ? ?      (Nothing is known)
COMMON A()
REM             ? + - - - ? - ?      (Known common array)
LET A(3,5) = 0
REM             ? + - - - ? - (E,E)  ("Parameters" known)
LET A = A(3,5)
REM             + + - - - + - (E,E)  (Scalar; default local)
CALL A
REM             X X - - X + - X      (Error -- can't be a sub)
```

The compiler has acquired knowledge about A in the following way. The COMMON statement tells it that there is an array whose name is A and that that array is in common. It still does not know how many dimensions A has (i.e., what its "parameters" are), and it does not know whether there is also a scalar named A. It knows that there is neither a function nor a subroutine named A. The first assignment statement tells it that A is a two-dimensional array (i.e., that its "parameter" list consists of two numeric expressions). It still knows nothing about a possible scalar A.

The second assignment tells it that there is a scalar named A. Nothing the compiler has seen heretofore tells it whether the scalar A is local or in common, so its residence defaults to local. At this point, there is no longer any uncertainty about A; there are no question marks in its vector.

The last statement is doubly illegal. Use of A as the name either of a scalar or of an array precludes its use as the name of a subroutine.

3.8 RESERVED WORDS

Words which have syntactic meaning in Virtual BASIC are reserved; they may not be used to denote program-defined objects. The reserved words are

AND, APPEND, AS, BASE, BEEP, CALL, CALLX, CHAIN, CLEAR, CLOSE, COMMON, DATA, DECLARE, DEF, DEFDBL, DEFINT, DEFSNG, DEFSTR, DIM, ELSE, ELSEIF, END, ENDIF, EQV, ERASE, ERROR, EXIT, EXTERNAL, FIELD, FOR, FUNCTION, GET, GOSUB, GOTO, IF, IMP, INPUT, LET, LINE, LSET, MID$, MOD, NEXT, NOT, ON, OPEN, OPTION, OR, OUTPUT, PRINT, PUT, RANDOMIZE, READ, REM, RESTORE, RESUME, RETURN, RSET, SPC, STEP, STOP, SUB, SWAP, TAB, THEN, TO, WEND, WHILE, WIDTH, WRITE, and XOR.

The names of library routines are not reserved. See the following section for rules governing their use.

3.9 NAMES OF LIBRARY ROUTINES

Names of Virtual BASIC library routines may be used to denote objects defined by the program. In any external routine, such names refer either to library routines or to program-defined objects; they never refer to both. The general rule is this: if the first use of such a name is consistent with a reference to the library routine, then every use must be consistent with a reference to the library routine, and every use will be so interpreted. If the first use of the name is inconsistent with a reference to the library routine, then each use must be consistent with the first, and none of them will be interpreted as a reference to the library routine.

The following program, therefore, is illegal.

```
L = LEN(S$)
FUNCTION LEN(S$)
LEN = 2
END FUNCTION
```

The first use of LEN is consistent with a reference to the library routine of that name. It is so interpreted. The second use is inconsistent with this interpretation; the program contains an error. (If the first line of the program were made its last, the program would be legal. LEN would refer to the program-defined function.)

4.0 PRODUCT-LEVEL DESCRIPTION

Virtual BASIC has important interfaces to CMML (the 180 Math Library) and to NOS/VE. CMML provides the bulk of the Virtual BASIC math library. NOS/VE provides operating system support services, notably linking, input/output, and task management.

The product call command for Virtual BASIC will be as defined in Section 2.2 of the SIS. Input and output formats will be as described in sections 2.3 and 3.3 of the SIS, except that the source file must conform to section 3 of this ERS as regards line numbers.

5.0 ERRORS

5.1 COMPILE-TIME_ERROR_PROCESSING

5.2 RUN-TIME_ERROR_PROCESSING

On entry to any routine, default exception handling is in effect. The Virtual BASIC programmer may elect to take control of exception handling by executing an on-error-statement.

Exception handling code selected by an on-error-statement need not handle exceptions. The code is characterized as exception handling by virtue of its selection, not by its function. It is possible to execute exception handling code that does nothing about the exception that caused it to be executed. It is also possible to execute exception handling code that is exactly the same code that caused the exception in the first place. While sound programming practices might be abetted if exception handling code were executed only as a result of an exception, Virtual BASIC does not require this to be so.

To assist a user's exception handling, each Virtual BASIC external routine supplies two integer-valued functions: ERL and ERR. ERL returns the value of the line number associated with the statement for which the routine's most recent exception was raised. The initial state of ERL is zero. ERR returns the ordinal of the current exception. A value of zero denotes either an initial or redeemed (see the resume statement) state of grace.

If an exception is raised while user-specified exception handling is in effect, an automatic branch to the exception handling code occurs. The exception is unresolved until the execution of a resume-statement. If a routine executes an exit-sub- or exit-function-statement while an exception is raised (i.e., at any time when ERR would return a non-zero value), the diagnostic information about the unresolved exception is saved in an error status queue, and an exception indicating an error in the called routine is raised at the site of the call in the calling routine. If a routine executes an end-statement while an exception is raised, or if a main-program terminates while an exception is unresolved, diagnostics describing the current exception and any others that may have queued are written to the standard error file. If an additional exception is raised while a previous exception in the same routine remains unresolved, the program is terminated.

If a nonfatal exception is raised while default exception handling is in effect, an informative message is written to the standard error file and the program resumes execution at the point of interruption. If a fatal exception is raised while default exception handling is in effect, diagnostic information describing the exception is stored in the error status queue. If the fatal exception was raised in a routine other than a main program, an exception indicating an error on the call is raised is raised at the site of the call. If the exception was raised in a main program, the diagnostic information in the error status queue is written to the standard error file and the program is terminated.

5.3 EXCEPTION CODES

(to be furnished)

6.0 APPENDIX A: GRAMMAR

6.0 APPENDIX A: GRAMMAR

What follows is the proceeds of extracting the grammar embedded in this document and processing it with the Bunter Parser Generator. It is included here to provide a cross reference to the embedded grammar.

```

! PAGE        3-3

        token = integer / real-number / quoted-string /
             unquoted-string / neme / integer-name /
             real-name / string-name / beginning-of-line
        integer = digit digit* /
             "&" "H" hex-digit hex-digit* /
             "&" "O"?  octal-digit octal-digit*
        hex-digit = digit / "A" / "B" / "C" / "D" / "E" / "F"
        digit = octal-digit / "8" / "9"
        octal-digit = "0" / "1" / "2" / "3" /
             "4" / "5" / "6" / "7"

! PAGE        3-4

        real-number = plain-real-number ("!"  / "#")?  /
             integer ("!"  / "#")
        plain-real-number = integer ("."  integer?
             decimal-exponent?  / decimal-exponent) /
             "."  integer decimal-exponent?
        decimal-exponent = "E" ("+" / "-")?  integer
        quoted-string = quote (non-quote / double-quote)* quote
        double-quote = quote quote
        quoted-string = """" (non-quote / """""")* """"

! PAGE        3-5

        unquoted-string = unquoted-string-char ((
             unquoted-string-char / " ")*
             unquoted-string-char)?
        name = letter name-char*
        name-char = letter / digit / "."
        integer-name = name "%"
        real-name = name ("!"  / "#")
        string-name = name "$"

! PAGE        3-7

        comment = "'" character* /
             statement-boundary "REM" character*
        statement-boundary = beginning-of-line line-number?  / ":"

! PAGE        3-10

        variable = numeric-variable / string-variable
        numeric-variable = numeric-array-element /
             numeric-scalar
        numeric-array-element = numeric-array-name subscript
        numeric-array-name = numeric-identifier
        subscript = "(" numeric-expression
             ("," numeric-expression)* ")"
        numeric-scalar = numeric-identifier
        numeric-identifier = name / integer-name / real-name
        string-variable = substring /

! PAGE        3-11
```

```

                whole-string-variable
            whole-string-variable = string-array-element /
                string-scalar
            string-array-element = string-array-name subscript
            string-array-name = string-identifier
            string-scalar = string-identifier
            string-identifier = string-name / name
            substring = mid-substring / colon-substring
            mid-substring = "MID$" "(" whole-string-variable ","
                first ("," length)?  ")"
            length = numeric-expression
            colon-substring = whole-string-variable "(" first
                ":" last ")"
            first = numeric-expression
            last = numeric-expression

! PAGE      3-12

            string-supplied-variable = "DATE$" / "TIME$"

! PAGE      3-13

            expression = numeric-expression / string-expression

! PAGE      3-15

            numeric-expression = first-eqv ("IMP" subs-eqv)*
            first-eqv = first-xor ("EQV" subs-xor)*
            subs-eqv = subs-xor ("EQV" subs-xor)*
            first-xor = first-or ("XOR" subs-or)*
            subs-xor = subs-or ("XOR" subs-or)*
            first-or = first-and ("OR" subs-and)*
            subs-or = subs-and ("OR" subs-and)*
            first-and = logical-not ("AND" logical-primary)*
            subs-and = logical-primary ("AND" logical-primary)*
            logical-not = "NOT"?  logical-primary
            logical-primary = relational-expression /
                arithmetic-expression
            relational-expression = string-expression relation
                string-expression (relation arithmetic-expression)* /
                arithmetic-expression relation arithmetic-expression
                (relation arithmetic-expression)*
            relation = "=" /
                "<" ">" / ">" "<" /
                ">" "=" / "=" ">" /
                "<" "=" / "=" "<" /
                ">" /
                "<"

! PAGE      3-16

            arithmetic-expression = first-mod (("+" / "-")
                subs-mod)*
            first-mod = first-sum ("MOD" subs-sum)*
            subs-mod = subs-sum ("MOD" subs-sum)*
            first-sum = first-id (("+" / "-") subs-id)*
```

```
! PAGE         3-3

            token = integer / real-number / quoted-string /
                 unquoted-string / name / integer-name /
                 real-name / string-name / beginning-of-line
            integer = digit digit* /
                 "&" "H" hex-digit hex-digit* /
                 "&" "O"?  octal-digit octal-digit*
            hex-digit = digit / "A" / "B" / "C" / "D" / "E" / "F"
            digit = octal-digit / "8" / "9"
            octal-digit = "0" / "1" / "2" / "3" /
                 "4" / "5" / "6" / "7"

! PAGE         3-4

            real-number = plain-real-number ("!"  / "#")?  /
                 integer ("!"  / "#")
            plain-real-number = integer ("."  integer?
                 decimal-exponent?  / decimal-exponent) /
                 "."  integer decimal-exponent?
            decimal-exponent = "E" ("+" / "-")?  integer
            quoted-string = quote (non-quote / double-quote)* quote
            double-quote = quote quote
            quoted-string = """" (non-quote / """""")* """"

! PAGE         3-5

            unquoted-string = unquoted-string-char ((
                 unquoted-string-char / " ")*
                 unquoted-string-char)?
            name = letter name-char*
            name-char = letter / digit / "."
            integer-name = name "%"
            real-name = name ("!"  / "#")
            string-name = name "$"

! PAGE         3-7

            comment = "!" character* /
                 statement-boundary "REM" character*
            statement-boundary = beginning-of-line line-number?  / ":"

! PAGE       3-10

            variable = numeric-variable / string-variable
            numeric-variable = numeric-array-element /
                 numeric-scalar
            numeric-array-element = numeric-array-name subscript
            numeric-array-name = numeric-identifier
            subscript = "(" numeric-expression
                 ("," numeric-expression)* ")"
            numeric-scalar = numeric-identifier
            numeric-identifier = name / integer-name / real-name
            string-variable = substring /

! PAGE       3-11
```

```

                    whole-string-variable
               whole-string-variable = string-array-element /
                    string-scalar
               string-array-element = string-array-name subscript
               string-array-name = string-identifier
               string-scalar = string-identifier
               string-identifier = string-name / name
               substring = mid-substring / colon-substring
               mid-substring = "MID$" "(" whole-string-variable ","
                    first ("," length)?  ")"
               length = numeric-expression
               colon-substring = whole-string-variable "(" first
                    ":" last ")"
               first = numeric-expression
               last = numeric-expression

! PAGE      3-12

               string-supplied-variable = "DATE$" / "TIME$"

! PAGE      3-13

               expression = numeric-expression / string-expression

! PAGE      3-15

               numeric-expression = first-eqv ("IMP" subs-eqv)*
               first-eqv = first-xor ("EQV" subs-xor)*
               subs-eqv = subs-xor ("EQV" subs-xor)*
               first-xor = first-or ("XOR" subs-or)*
               subs-xor = subs-or ("XOR" subs-or)*
               first-or = first-and ("OR" subs-and)*
               subs-or = subs-and ("OR" subs-and)*
               first-and = logical-not ("AND" logical-primary)*
               subs-and = logical-primary ("AND" logical-primary)*
               logical-not = "NOT"?  logical-primary
               logical-primary = relational-expression /
                    arithmetic-expression
               relational-expression = string-expression relation
                    string-expression (relation arithmetic-expression)* /
                    arithmetic-expression relation arithmetic-expression
                    (relation arithmetic-expression)*
               relation = "=" /
                    "<" ">" / ">" "<" /
                    ">" "=" / "=" ">" /
                    "<" "=" / "=" "<" /
                    ">" /
                    "<"

! PAGE      3-16

               arithmetic-expression = first-mod (("+" / "-")
                    subs-mod)*
               first-mod = first-sum ("MOD" subs-sum)*
               subs-mod = subs-sum ("MOD" subs-sum)*
               first-sum = first-id (("+" / "-") subs-id)*
```

```
          subs-sum = subs-id (("+" / "-") subs-id)*
          first-id = first-prod ("\" subs-prod)*
          subs-id = subs-prod ("\" subs-prod)*
          first-prod = negation (("*" / "/") exponentiation)*
          subs-prod = exponentiation (("*" / "/")
               exponentiation)*
          negation = "-"?  exponentiation
          exponentiation = numeric-primary ("^" numeric-primary)*
          numeric-primary = numeric-variable /
               numeric-constant /
               numeric-expression-function-ref /
               numeric-function-ref /
               "(" numeric-expression ")"
          numeric-constant = integer / real-number
          numeric-expression-function-ref = numeric-identifier
               exp-fn-actual-param-list?
          exp-fn-actual-param-list = "(" expression
               ("," expression)* ")"
          numeric-function-ref = numeric-identifier
               actual-parameter-list?

! PAGE    3-17

          string-expression = string-primary
               ("+" string-primary)*
          string-primary = string-variable / string-function-ref /
               string-expression-function-ref /
               "(" string-expression ")"
          string-expression-function-ref = string-identifier
               exp-fn-actual-param-list?
          string-function-ref = string-identifier
               actual-parameter-list?
          declarative-statement = common-statement /
               expression-function-definition /
               function-declaration-statement /
               option-base-statement /
               subroutine-declaration-statement /
               type-declaration-statement

! PAGE    3-18

          common-statement = "COMMON" common-list
          common-list = common-list-item ("," common-list-item)*
          common-list-item = identifier ("(" ")")?

! PAGE    3-19

          expression-function-definition = "DEF" (
               numeric-expression-function /
               string-expression-function )
          numeric-expression-function = numeric-identifier
               ex-fun-formal-parameter-list?  "=" numeric-expression
          string-expression-function = string-identifier
               ex-fun-formal-parameter-list?  "=" string-expression
          ex-fun-formal-parameter-list = "(" identifier
               ("," identifier) ")"
          function-declaration-statement = "DECLARE"
```

```
          "EXTERNAL"?  "FUNCTION" function-name-list
     function-name-list = function-name
          ("," function-name)*

! PAGE       3-20

     option-base-statement = "OPTION" "BASE" ("0"/"1")
     subroutine-declaration-statement = "DECLARE"
          "EXTERNAL" "SUB" subroutine-name-list
     subroutine-name = subroutine-name
          ("," subroutine-name)*
     type-declaration-statement = defint-statement /

! PAGE       3-21

          defreal-statement / defstr-statement
     defint-statement = "DEFINT" letter-list
     defreal-statement = ("DEFSNG" / "DEFDBL")
          letter-list
     defstr-statement = "DEFSTR" letter-list
     letter-list = letter-list-item (","
          letter-list-item)*
     letter-list-item = letter-range / letter
     letter-range = letter "-" letter
     letter = "A" / "B" / "C" / "D" / "E" /
          "F" / "G" / "H" / "I" / "J" / "K" /
          "L" / "M" / "N" / "O" / "P" / "Q" /
          "R" / "S" / "T" / "U" / "V" / "W" /
          "X" / "Y" / "Z"

! PAGE       3-22

     assignment-statement = let-statement /
          swap-statement
     let-statement = "LET"?  (numeric-assignment /
          string-assignment)
     numeric-assignment = numeric-variable "="
          numeric-expression
     string-assignment = string-left-hand-side "="
          string-expression
     string-left-hand-side =
          string-supplied-variable / string-variable
     swap-statement = "SWAP" (numeric-variable ","
          numeric-variable / string-variable ","
          string-variable)

! PAGE       3-23

     control-statement = call-statement /
          callx-statement /
          chain-statement /
          end-statement /
          error-statement /
          exit-function-statement /
          exit-sub-statement /
          gosub-statement /
          goto-statement /
```

```
                    on-error-statement /
                    on-gosub-statement /
                    on-goto-statement /
                    resume-statement /
                    return-statement /
                    run-statement /
                    stop-statement
               call-statement = "CALL" subroutine-name
                    actual-parameter-list?
               actual-parameter-list = "(" actual-parameter
                    ("," actual-parameter)* ")"

! PAGE     3-24

               actual-parameter = expression / actual-array
               actual-array = identifier "(" ","* ")"
               identifier = name / integer-name / real-name /
                    string-name
               callx-statement = "CALLX" external-routine-name
                    external-parameter-list?
               external-routine-name = letter (letter / digit)*
               external-parameter-list = "(" external-parameter
                    ("," external-parameter)* ")"
               external-parameter = expression / external-array
               external-array = numeric-identifier "(" ","* ")"

! PAGE     3-25

               chain-statement = "CHAIN" file-name
               file-name = string-expression
               end-statement = "END"
               error-statement = "ERROR" numeric-expression

! PAGE     3-26

               exit-function-statement = "EXIT" "FUNCTION"
               exit-sub-statement = "EXIT" "SUB"
               gosub-statement = "GOSUB" line-number
               goto-statement = "GOTO" line-number

! PAGE     3-27

               on-error-statement = "ON" "ERROR" "GOTO" line-number
               on-gosub-statement = "ON" numeric-expression
                    "GOSUB" line-number ("," line-number)*

! PAGE     3-28

               on-goto-statement = "ON" numeric-expression
                    "GOTO" line-number ("," line-number)*
               resume-statement = "RESUME" (
                    "NEXT" /
                    line-number /
                    "0"?  )

! PAGE     3-29
```

```
          return-statement = "RETURN"
          run-statement = "RUN" string-expression
          stop-statement = "STOP"
          io-statement = close-statement /
               field-statement /
               get-statement /
               input-statement /
               line-input-statement /
               lprint-statement /
               lprint-using-statement /
               lset-statement /

! PAGE      3-30

               open-statement /
               print-statement /
               print-using-statement /
               put-statement /
               rset-statement /
               width-statement /
               write-statement /
               beep-statement

! PAGE      3-31

          open-statement = "OPEN" (open1 / open2)
          open1 = file-name ("FOR" io-mode)?  "AS"
               "#"?  numeric-expression
               ("LEN" "=" numeric-expression)?
          open2 = string-expression "," "#"?
               numeric-expression "," file-name
               ("," numeric-expression)?
          io-mode = "INPUT" / "OUTPUT" / "APPEND"

! PAGE      3-33

          close-statement = "CLOSE" file-number-list?
          file-number-list = "#"?  numeric-expression
               ("," "#"?  numeric-expression)*

! PAGE      3-34

          field-statement = "FIELD" "#"?  numeric-expression ","
               field-list
          field-list = field-item ("," field-item)*
          field-item = numeric-expression "AS"
               whole-string-variable

! PAGE      3-36

          get-statement = "GET" "#"?  numeric-expression
               ("," numeric-expression)?

! PAGE      3-37

          input-statement = "INPUT" ";"?
               ("#" numeric-expression ",")?
```

```
                    (string-constant (";"/","))?
                    variable-list
                variable-list = variable-list-item
                    ("," variable-list-item)*
                variable-list-item = numeric-variable /
                    whole-string-variable

     ! PAGE     3-39

                line-input-statement = "LINE" "INPUT" ";"?
                    ("#" numeric-expression ",")?
                    (string-expression ";")?
                    whole-string-variable

     ! PAGE     3-40

                lprint-statement = "LPRINT" print-list?
                lprint-using-statement = "LPRINT" "USING"
                    string-expression ";" print-using-list ";"?
                lset-statement = "LSET" string-variable "="
                    string-expression
                rset-statement = "RSET" string-name "="
                    string-expression

     ! PAGE     3-41

                print-statement = "PRINT" ("#" numeric-expression ",")?
                    print-list?
                print-list = print-list-item
                    ((","/";")?  print-list-item)* (";"/",")?
                print-list-item = format-function / expression
                format-function = "SPC" "(" numeric-expression ")" /
                    "TAB" "(" numeric-expression ")"

     ! PAGE     3-43

                print-using-statement = "PRINT"
                    ("#" numeric-expression ",")?
                    "USING" string-expression ";"
                    print-using-list
                print-using-list = expression ((","/";")?  expression)*
                    (","/";")?

     ! PAGE     3-46

                put-statement = "PUT" "#"?  numeric-expression
                    ("," numeric-expression)?
                width-statement = "WIDTH"
                    (("#"?  numeric-expression / file-name) ",")?
                    numeric-expression

     ! PAGE     3-47

                write-statement = "WRITE" ("#" numeric-expression ",")?
                    write-list?
                write-list = expression ((","/";") expression)*
                beep-statement = "BEEP"
```

```

! PAGE      3-48

        data-initialization-statement = data-statement /
             read-statement / restore-statement
        data-statement = "DATA" datum ("," datum)*
        datum = unquoted-string / quoted-string
        read-statement = "READ" variable-list

! PAGE      3-49

        restore-statement = "RESTORE" line-number?
        miscellaneous-executable-statement =
             clear-statement /
             dim-statement /
             erase-statement /
             randomize-statement

! PAGE      3-50

        clear-statement = "CLEAR"
        dim-statement = "DIM" array-declaration (","
             array-declaration)*
        array-declaration = array-name "(" dimension-bounds
             ("," dimension-bounds)* ")"
        array-name = identifier
        dimension-bounds = (numeric-expression ":")?
             numeric-expression

! PAGE      3-51

        erase-statement = "ERASE" array-name (
             "," array-name)*
        randomize-statement = "RANDOMIZE" numeric-expression?

! PAGE      3-52

        block = block-element
             (statement-boundary block-element)*
        block-element = unstructured-statement /
             line-if-statement /
             for-block /
             if-block /
             internal-function /
             internal-subroutine /
             while-block
        unstructured-statement = assignment-statement /
             control-statement /
             declarative-statement /
             io-statement /
             data-initialization-statement /
             miscellaneous-executable-statement
        line-if-statement = "IF" numeric-expression ("THEN"
             (line-block / line-number) / "GOTO" line-number)
             ("ELSE" (line-block / line-number))?

! PAGE      3-53
```

```
                    line-block = block-element (":" block-element)*
                    for-block = for-statement
                         delimited-optional-blk next-statement
                    delimited-optional-blk = statement-boundary
                         (block statement-boundary)?
                    for-statement = "FOR" control-variable "="
                         numeric-expression "TO" numeric-expression
                         ("STEP" numeric-expression)?
                    control-variable = numeric-scalar
                    next-statement = "NEXT" control-variable-list?
                    control-variable-list = control-variable
                         ("," control-variable)?

      ! PAGE      3-54

                    if-block = "IF" if-body
                    if-body = numeric-expression "THEN" consequent if-tail
                    if-tail = "ELSEIF" if-body / ("ELSE" alternate)?  "ENDIF"

      ! PAGE      3-55

                    consequent = delimited-optional-blk
                    alternate = delimited-optional-blk
                    while-block = "WHILE" numeric-expression while-body "WEND"
                    while-body = delimited-optional-blk

      ! PAGE      3-57

                    source-deck = line-number?  external-routine
                         (statement-boundary external-routine)*
                    line-number = integer
                    external-routine = main-program /
                         external-function / external-subroutine

      ! PAGE      3-58

                    main-program = block
                    external-function = external-function-statement
                         delimited-block end-function-statement
                    delimited-block = statement-boundary block
                         statement-boundary
                    external-subroutine = external-sub-statement
                         delimited-optional-blk end-sub-statement
                    external-function-statement = "EXTERNAL" "FUNCTION"
                         function-name formal-parameter-list?
                    external-sub-statement = "EXTERNAL" "SUB"
                         subroutine-name formal-parameter-list?
                    formal-parameter-list = "(" formal-parameter
                         ("," formal-parameter)* ")"
                    end-function-statement = "END" "FUNCTION"
                    end-sub-statement = "END" "SUB"
                    formal-parameter = formal-scalar / formal-array

      ! PAGE      3-59

                    formal-scalar = identifier
```

```
formal-array = identifier "(" ","* ")"
subroutine-name = name
function-name = string-function-name /
     numeric-function-name
string-function-name = string-identifier
numeric-function-name = numeric-identifier
internal-function = internal-function-statement
     statement-boundary internal-block statement-boundary
     end-function-statement
internal-subroutine = internal-sub-statement
     statement-boundary (internal-block
     statement-boundary)?
     end-sub-statement
internal-function-statement = "FUNCTION" function-name
     formal-parameter-list?
internal-sub-statement = "SUB" subroutine-name
     formal-parameter-list?
internal-block = internal-block-element
     (statement-boundary internal-block-element)*
internal-block-element = unstructured-statement /
     line-if-statement /
     for-block /
     if-block /
     while-block
```

```
        30T
!       17T   18T   35T
"       25T   25T
""            25T
#       17T   18T   35T  313T  315T  323T
       324T  328T  336T  342T  353T  369T
       380T  388T  391T  396T
$       36T
%       34T
&        8T    9T
'       40T
(       51T   67T   70T  127T  131T  142T
       158T  169T  238T  244T  250T  253T
       374T  375T  423T  505T  514T
)       52T   68T   71T  127T  132T  142T
       158T  170T  239T  244T  251T  253T
       374T  375T  424T  506T  514T
*      118T  119T
+       22T  110T  114T  115T  139T
,       52T   67T   68T  132T  157T  170T
       174T  182T  192T  214T  215T  239T
       244T  251T  253T  273T  278T  315T
       316T  317T  324T  328T  330T  337T
       342T  343T  346T  353T  369T  372T
       372T  380T  383T  384T  389T  391T
       396T  398T  405T  421T  424T  432T
       469T  506T  514T
-       22T  110T  114T  115T  121T  195T
.       19T   21T   33T
/      118T  119T
0       12T  178T  262T
1       12T  178T
2       12T
3       12T
4       13T
5       13T
6       13T
7       13T
8       11T
9       11T
:       42T   71T  426T  458T
;      341T  343T  352T  354T  361T  361T
       372T  372T  381T  383T  384T  398T
<      102T  102T  104T  104T  106T
=      101T  103T  103T  104T  104T  166T
       168T  208T  210T  314T  362T  364T
       463T
>      102T  102T  103T  103T  105T
A       10T  196T
AND           92T   93T
APPEND       318T
AS           312T  331T
B       10T  196T
BASE         178T
BEEP         399T
C       10T  196T
CALL         236T

CALLX         247T
CHAIN         257T
CLEAR         420T
CLOSE         322T
COMMON        156T
D        10T  196T
DATA          405T
DATE$          77T
DECLARE       171T  179T
DEF           162T
DEFDBL        189T
DEFINT        188T
DEFSNG        189T
DEFSTR        191T
DIM           421T
E        10T   22T  196T
ELSE          454T  475T
ELSEIF        475T
END           259T  507T  508T
ENDIF         475T
EQV            86T   87T
ERASE         431T
ERROR         260T  271T
EXIT          264T  265T
EXTERNAL            172T  180T  501T  503T
F        10T  197T
FIELD         328T
FOR           312T  463T
FUNCTION            172T  264T  501T  507T
        527T
G       197T
GET           336T
GOSUB         266T  273T
GOTO          267T  271T  278T  453T
H         8T  197T
I       197T
IF            452T  473T
IMP            85T
INPUT         318T  341T  352T
J       197T
K       197T
L       198T
LEN           314T
LET           206T
LINE          352T
LPRINT        359T  360T
LSET          362T
M       198T
MID$           67T
MOD           112T  113T
N       198T
NEXT          280T  467T
NOT            94T
O         9T  198T
ON            271T  272T  277T
OPEN          311T
OPTION        178T

OR              90T   91T
OUTPUT         318T
P       198T
PRINT          369T  379T
PUT            388T
Q       198T
R       199T
RANDOMIZE            433T
READ           407T
REM             41T
RESTORE        411T
RESUME         279T
RETURN         286T
RSET           364T
RUN            287T
S       199T
SPC            374T
STEP           465T
STOP           288T
SUB            180T  265T  503T  508T  529T
SWAP           214T
T       199T
TAB            375T
THEN           452T  474T
TIME$           77T
TO             464T
U       199T
USING          360T  381T
V       199T
W       199T
WEND           481T
WHILE          481T
WIDTH          390T
WRITE          396T
X       200T
XOR             88T   89T
Y       200T
Z       200T
\       116T  117T
^       122T
actual-array (244)       243N
actual-parameter (243)         238N  239N
actual-parameter-list (238)          134N
        146N  237N
alternate (480)          475N
arithmetic-expression (110)           96N
          98N   99N   99N  100N
array-declaration (423)        421N  422N
array-name (425)         423N  431N  432N
assignment-statement (204)     446N
beep-statement (399)     307N
beginning-of-line          6P   42P
block (437)        462N  494N  497N
block-element (439)      437N  438N  458N
        458N
call-statement (236)     220N
callx-statement (247)          221N
```

```
chain-statement (257)  222N
character  40P  41P
clear-statement (420)  413N
close-statement (322)  289N
colon-substring (70)  66N
comment (40)
common-list (157)  156N
common-list-item (158)  157N  157N
common-statement (156)  147N
consequent (479)  474N
control-statement (220)  447N
control-variable (466)  463N  468N
    469N
control-variable-list (468)  467N
data-initialization-statement (403)
    450N
data-statement (405)  403N
datum (406)  405N  405N
decimal-exponent (22)  20N  20N
    21N
declarative-statement (147)  448N
defint-statement (188)  183N
defreal-statement (189)  187N
defstr-statement (191)  187N
delimited-block (497)  496N
delimited-optional-blk (461)  460N
    479N  480N  482N  500N
digit (11)  7N  7N  10N  33N
    249N
dim-statement (421)  414N
dimension-bounds (426)  423N  424N
double-quote (24)  23N
end-function-statement (507)  496N
    522N
end-statement (259)  223N
end-sub-statement (508)  500N  526N
erase-statement (431)  415N
error-statement (260)  224N
ex-fun-formal-parameter-list (169)
    166N  168N
exit-function-statement (264)  225N
exit-sub-statement (265)  226N
exp-fn-actual-param-list (131)  130N
    144N
exponentiation (122)  118N  119N  120N
    121N
expression (81)  131N  132N  243N
    252N  373N  383N  383N  398N  398N
expression-function-definition (162)
    148N
external-array (253)  252N
external-function (495)  490N
external-function-statement (501)
    495N
external-parameter (252)  250N  251N
external-parameter-list (250)  248N
external-routine (489)  486N  487N
external-routine-name (249)  247N
external-sub-statement (503)  499N
external-subroutine (499)  490N
field-item (331)  330N  330N
field-list (330)  329N
field-statement (328)  290N
file-name (258)  257N  312N  316N
    391N
file-number-list (323)  322N
first (72)  68N  70N
first-and (92)  90N
first-eqv (86)  85N
first-id (116)  114N
first-mod (112)  110N
first-or (90)  88N
first-prod (118)  116N
first-sum (114)  112N
first-xor (88)  86N
for-block (459)  441N  535N
for-statement (463)  459N
formal-array (514)  509N
formal-parameter (509)  505N  506N
formal-parameter-list (505)  502N
    504N  528N  530N
formal-scalar (513)  509N
format-function (374)  373N
function-declaration-statement (171)
    149N
function-name (516)  173N  174N  502N
    527N
function-name-list (173)  172N
get-statement (336)  291N
gosub-statement (266)  227N
goto-statement (267)  228N
hex-digit (10)  8N  8N
identifier (245)  158N  169N  170N
    244N  425N  513N  514N
if-block (473)  442N  536N
if-body (474)  473N  475N
if-tail (475)  474N
input-statement (341)  292N
integer (7)  4N  18N  19N  19N
    21N  22N  128N  488N
integer-name (34)  5N  54N  245N
internal-block (531)  521N  524N
internal-block-element (533)  531N
    532N
internal-function (520)  443N
internal-function-statement (527)
    520N
internal-sub-statement (529)  523N
internal-subroutine (523)  444N
io-mode (318)  312N
io-statement (289)  449N
last (73)  71N
length (69)  68N
let-statement (206)  204N
letter (196)  32N  33N  194N  195N
    195N  249N  249N
letter-list (192)  188N  190N  191N
letter-list-item (194)  192N  193N
letter-range (195)  194N
line-block (458)  453N  454N
line-if-statement (452)  440N  534N
line-input-statement (352)  293N
line-number (488)  42N  266N  267N
    271N  273N  273N  278N  278N  281N
    411N  453N  453N  454N  486N
logical-not (94)  92N
logical-primary (95)  92N  93N  93N
    94N
lprint-statement (359)  294N
lprint-using-statement (360)  295N
lset-statement (362)  296N
main-program (494)  489N
mid-substring (67)  66N
miscellaneous-executable-statement (412)
    451N
name (32)  5N  34N  35N  36N
    54N  65N  245N  515N
name-char (33)  32N
negation (121)  118N
next-statement (467)  460N
non-quote  23P  25P
numeric-array-element (49)  47N
numeric-array-name (50)  49N
numeric-assignment (208)  206N
numeric-constant (128)  124N
numeric-expression (85)  51N  52N
    69N  72N  73N  81N  127N  166N
    209N  260N  272N  277N  313N  314N
    316N  317N  323N  324N  328N  331N
    336N  337N  342N  353N  369N  374N
    375N  380N  388N  389N  391N  392N
    396N  426N  427N  433N  452N  464N
    464N  465N  474N  481N
numeric-expression-function (165)
    163N
numeric-expression-function-ref (129)
    125N
numeric-function-name (519)  517N
numeric-function-ref (133)  126N
numeric-identifier (54)  50N  53N
    129N  133N  165N  253N  519N
numeric-primary (123)  122N  122N
numeric-scalar (53)  48N  466N
numeric-variable (47)  46N  123N
    208N  214N  215N  347N
octal-digit (12)  9N  9N  11N
on-error-statement (271)  229N
on-gosub-statement (272)  230N
on-goto-statement (277)  231N
open-statement (311)  300N
openl (312)  311N
```

```
open2 (315)        311N
option-base-statement (178)         150N
plain-real-number (19)        17N
print-list (371)        359N  370N
print-list-item (373)         371N  372N
print-statement (369)         301N
print-using-list (383)        361N  382N
print-using-statement (379)         302N
put-statement (388)     303N
quote        23P   23P   24P   24P
quoted-string (23)       4N  406N
randomize-statement (433)     416N
read-statement (407)    404N
real-name (35)     6N   54N  245N
real-number (17)         4N  128N
relation (101)     97N   98N   99N  100N
relational-expression (97)     95N
restore-statement (411)       404N
resume-statement (279)        232N
return-statement (286)        233N
rset-statement (364)    304N
run-statement (287)     234N
source-deck (486)
statement-boundary (42)        41N  438N
      461N  462N  487N  497N  498N  521N
      521N  524N  525N  532N
stop-statement (288)    235N
string-array-element (62)      60N
string-array-name (63)         62N
string-assignment (210)       207N
string-constant         343P
string-expression (138)        81N   97N
       98N  142N  168N  211N  258N  287N
      315N  354N  361N  363N  365N  381N
string-expression-function (167)    164N
string-expression-function-ref (143)
      141N
string-function-name (518)    516N
string-function-ref (145)     140N
string-identifier (65)         63N   64N
      143N  145N  167N  518N
string-left-hand-side (212)         210N
string-name (36)         6N   65N  246N
      364N
string-primary (140)    138N  139N
string-scalar (64)      61N
string-supplied-variable (77)       213N
string-variable (55)     46N  140N  213N
      215N  216N  362N
subroutine-declaration-statement (179)
      151N
subroutine-name (181)         181N  182N
      236N  504N  529N
subroutine-name-list          180P
subs-and (93)      90N   91N   91N
subs-eqv (87)      85N
subs-id (117)     114N  115N  115N
subs-mod (113)     111N
subs-or (91)        88N   89N   89N
subs-prod (119)         116N  117N  117N
subs-sum (115)     112N  113N  113N
subs-xor (89)       86N   87N   87N
subscript (51)      49N   62N
substring (66)      55N
swap-statement (214)    205N
token (4)
type-declaration-statement (183)    152N
unquoted-string (29)       5N  406N
unquoted-string-char          29P   30P
       31P
unstructured-statement (446)        439N
      533N
variable (46)
variable-list (345)     344N  407N
variable-list-item (347)      345N  346N
while-block (481)       445N  537N
while-body (482)        481N
whole-string-variable (60)     59N   67N
       70N  332N  348N  355N
width-statement (390)         305N
write-list (398)        397N
write-statement (396)         306N
```

The pseudo-terminals of the grammar are beginning-of-line, character, non-quote, quote, string-constant, subroutine-name-list and unquoted-string-char.