# BASIC Compiler Language

Lorenz Wiest | Version 2015-03-25

Document History

| | |
|---|---|
| LW 2015-03-21 | Revision |
| LW 2015-03-05 | Updating numbers |
| LW 2014-05-12 | Adding DEF FN |
| LW 2013-12-16 | Cosmetic changes |
| LW 2013-07-07 | PRINT: Spaces as expression separators |
| LW 2013-05-27 | Revision |
| LW 2013-04-11 | Revision |

# Table of Contents

# 1. Introduction

This document describes the implemented BASIC language of the BASIC Compiler project. The implemented BASIC language is oriented at Microsoft BASIC.

## 1.1 Syntax Notation

- These *words* are placeholders that must be filled in by the programmer.
- *[]* Items in square brackets are optional.
- *{}* Items in curly braces indicate a set of choices.
- *|* A vertical bar separates choices within curly braces.
- *** The preceeding item can be repeated zero, one, or more times.

# 2. Basics

This section describes line format, numbers, strings, their operators, variables, and arrays.

## 2.1 Line Format

A BASIC program is composed of lines of code. Each line of code starts with a line number, followed by one or more statements separated by a colon (`:`). The general format is:

*lineNumber statement[*`:`*statement]**

- A *lineNumber* is in the range of 0 to 99999.
- A line of code contains up to 255 characters.
- Blank lines of code are ignored.
- All lines of code are sorted by their line number in increasing order.
- If there are two lines of code with the same line number, then the first line of code is ignored.

## 2.2 Numbers

- Numbers are represented internally by IEEE 754-1985 float values.
- Number constants match the regular expression `[-+]?([0-9]+(\.[0-9]*)?|\.[0-9]++)([eE][-+]?[0-9]+)?`.
- The maximum positive number is 3.402823e+38.
- The maximum negative number is -3.402823e+38.
- Numbers 0 and -0 are identical.

## 2.3 Operators for Numbers

The following types of operators can be applied to numbers (in descending order of priority):

- Arithmetic Operators
- Relational Operators
- Logical Operators

### 2.3.1 Arithmetic Operators

The arithmetic operators are (in descending order of priority):

| Operator | Description | Example | Result | Priority |
|---|---|---|---|---|
| `^` | Power | `2^3` | `8` | 6 |
| `-` | Unary Minus | `-3` | `-3` | 5 |
| `*` | Multiplication | `2*3` | `6` | 4 |
| `/` | Division | `6/3` | `2` | 4 |
| `\` | Integer Division | `12\5` | `2` | 3 |
| `MOD` | Modulo | `6 MOD 4` | `2` | 2 |
| `+` | Addition | `2+3` | `5` | 1 |
| `-` | Subtraction | `2-3` | `-1` | 1 |

Division `/`

- If the denominator is 0 then `Division by zero` is printed and the result is infinity with the sign of the numerator.

Integer Division `\`

- The arguments must be in the range of -32768 to +32767.
- The quotient is truncated to an integer value.
- If the denominator is 0 then `Division by zero` is printed and the result is infinity with the sign of the numerator.

Integer remainder `MOD`

- The arguments must be in the range of -32768 to +32767.
- If the denominator is 0 then `Division by zero` is printed and the result is infinity with the sign of the numerator.

### 2.3.2 Relational Operators

The relational operators are:

- `<` Less than
- `<=` Less or equal than
- `=` Equals
- `<>` Unequal to
- `>=` Greater or equal than
- `>` Greater than

The result of a relational operator is either -1 (true) or 0 (false).

### 2.3.3 Logical Operators

The logical operators are:

- `AND` And
- `OR` Or
- `XOR` Exclusive Or
- `NOT` Not

Logical operators convert the argument(s) to signed 16-bit integer values in the range of -32768 to 32767, perform the logical operation, and return the result as a signed 16-bit integer value. If the argument values are not in the signed 16-bit integer value range, then an error occurs.

| Operation | Result |
|---|---|
| `0 AND 0` | `0` |
| `0 AND 1` | `0` |
| `1 AND 0` | `0` |
| `1 AND 1` | `1` |

| Operation | Result |
|---|---|
| `0 OR 0` | `0` |
| `0 OR 1` | `1` |
| `1 OR 0` | `1` |
| `1 OR 1` | `1` |

| Operation | Result |
|---|---|
| `0 XOR 0` | `0` |
| `0 XOR 1` | `1` |
| `1 XOR 0` | `1` |
| `1 XOR 1` | `0` |

| Operation | Result |
|---|---|
| `NOT 0` | `1` |
| `NOT 1` | `0` |

| Example | Result |
|---|---|
| `1 AND 1` | `1` |
| `7 AND 3` | `3` |
| `6 AND 3` | `2` |
| `1 OR 1` | `1` |
| `7 OR 3` | `7` |
| `6 OR 3` | `7` |
| `1 XOR 1` | `0` |
| `7 XOR 3` | `4` |
| `6 XOR 3` | `5` |
| `NOT 1` | `-2` |
| `NOT 7` | `-8` |
| `NOT 3` | `-4` |

## 2.4 Strings

- Strings contain up to 255 ASCII characters.
- String constants are enclosed in double quotes (`"`).

## 2.5 Operators for Strings

The following types of operators can be applied to strings (in descending order of priority):

- Concatenation Operator
- Relational Operators

### 2.5.1 Concatenation Operator

The string concatenation operator is `+`.

| Operation | Result |
|---|---|
| `"ABC"+"DEF"` | `"ABCDEF"` |

### 2.5.2 Relational Operators

The relational operators for strings are:

- `<` Less than
- `<=` Less or equal than
- `=` Equals
- `<>` Unequal to
- `>=` Greater or equal than
- `>` Greater than

The result of a relational operator is either -1 (true) or 0 (false).

Relational operators compare both strings character for character by their ASCII codes. Strings are *equal* if the ASCII codes of both strings are the same. If during the comparison a character of the first string has a lower ASCII code than the second string, then the first string is *less than* the second string. If during the comparison the end of the first string is reached before the end of the second string, then the first string is *less than* the second string, too.

| Operation | Result |
|---|---|
| `"ABC"="ABC"` | `-1` (true) |
| `"ABC"="ABD"` | `0` (false) |
| `"ABC"<"ABD"` | `-1` (true) |
| `"ABC"<"ABCD"` | `-1` (true) |

## 2.6 Variables

- A variable represents either a number, a string, or an array of numbers or strings.
- Each variable has a name. The name indicates the type of the variable:

| Variable represents | Variable Name (Regex notation) | Examples |
|---|---|---|
| Number | `[A-Z][A-Z0-9\.]*` | `A` |
| String | `[A-Z][A-Z0-9\.]*\$` | `A$` |
| Array of numbers | `[A-Z][A-Z0-9\.]*\(...\)` | `A(5), A(2,2)` |
| Array of strings | `[A-Z][A-Z0-9\.]*\$\(...\)` | `A$(3), A$(2,3)` |

- Variable names may have any number of characters
- Variables names must be different from reserved words for statements, functions, and operators.
- Variable names `A`, `A$`, `A(1)`, `A$(1)` represent four distinct variables.
- Number variables and number array variables are initially set to 0.
- String variables and string array variables are initially set to the empty string (`""`).

## 2.7 Arrays

- Memory for array variables must be allocated with the `DIM` statement.
- An array variable has 1 or 2 indexes.
- The minimum array variable index value is 0, the maximum array variable index value depends on the size of the array (see `DIM` statement), but is less than 32768.
- Array variable index values are rounded to integer values.

# 3. Statements

This section lists all statements of the implemented BASIC language.

## 3.1 DATA

**Format:** **DATA** *constant[,constant]**

**Description:** Stores number and string constants. String constants that contain commas (**,**), colons (**:**), or leading or trailing spaces must be enclosed in double quotes (**"**). Constants stored in **DATA** statements are retrieved by **READ** statements in order by line number. **DATA** statements can be placed anywhere in a program.

**Example:**

```
10 FOR I=1 TO 3
20 READ A$
30 PRINT A$
40 NEXT I
50 DATA PARIS,LONDON,ROME

PARIS
LONDON
ROME
```

**See also:** **READ**
**RESTORE**

## 3.2 DEF FN

**Format:** **DEF FN***name***(***parameter[,parameter]**)=expression*

**Description:** Defines a user-defined function. The function name is **FN** followed by *name*, where *name* must be a valid variable name. A function has one or more *parameter*s that are replaced with the actual values when the function is called. The *expression* evaluates the value of the function. It can contain variables and parameters.
A user-defined function can define a number function or a string function. The type of its *name* must be the same as the type of its *expression*.
A user-defined function must fit in one line of code.
A user-defined function must be defined before it can be called.
A user-defined function of the same name cannot be defined twice.

**Example:**

```
10 DEF FNA(X)=X*X*X
20 PRINT FNA(2)

 8

10 DEF FNMULT(X,Y) = X * Y
20 PRINT FNMULT(2,3)

 6
```

```
10 DEF FNFIRST$(A$)=LEFT(A$,1)
20 PRINT FNFIRST$("HELLO")

H
```

## 3.3 DIM

**Format:** `DIM` *arrayVariable[,arrayVariable]**

**Description:** Allocates memory for one or more array variables.

**Example:**

```
10 DIM SQUARE(3)
20 FOR I=0 TO 2
30 SQUARE(I)=I*I
40 NEXT I
50 FOR I=0 TO 2
60 PRINT I,SQUARE(I)
70 NEXT I

 0            0
 1            1
 2            4
```

## 3.4 END

**Format:** **END**

**Description:** Ends the program. The **END** statment at the end of a program is optional.

**Example:** `10 IF A=1 THEN END ELSE RETURN`

## 3.5 FOR...NEXT

**Format:** **FOR** *numberVariable*=*startNumExpression* **TO** *endNumExpression* *[***STEP** *stepNumExpression]*

...

**NEXT** *[numberVariable[,numberVariable]*]*

**Description:** Executes a sequence of statements repeatedly with *numberVariable* acting as a counter. First, *numberVariable* is set to the result of *startNumExpression* and the results of *endNumExpression* and *stepNumExpression* are calculated. If **STEP** is omitted then *stepNumExpression* is 1. Then the statements between **FOR** and **NEXT** are executed. After that the value of *numberVariable* is increased by the result of *stepNumExpression*. If the updated value of *numberVariable* is smaller or equal to the previously computed result of *endNumExpression* then the statements between **FOR** and **NEXT** are executed again, otherwise program execution continues at the statement after **NEXT**. The statements between **FOR** and **NEXT** are skipped altogether if *startNumExpression* * **SGN(***stepNumExpression***)** > *endNumExpression* * **SGN(***stepNumExpression***)**. FOR-NEXT loops may be nested, each loop must have its own counter variable *numberVariable*. The *numberVariable* in **NEXT**

statements is optional; program execution will loop back to the most recent FOR statement.

**Example:**

```
10 FOR I=1 TO 3
20 PRINT I,I*I
30 NEXT I

 1             1
 2             4
 3             9

10 FOR I=1 TO 5 STEP 2
20 PRINT I,I*I
30 NEXT I

 1             1
 3             9
 5             25

10 FOR I=1 TO 3
20 FOR J=2 TO 4
30 PRINT I*J;
40 NEXT J
50 NEXT I

 2  3  4  2  4  6  3  6  9

10 ST=3
20 FOR I=1 TO 4 STEP ST
30 ST=1
40 PRINT I
50 NEXT

 1
 4

10 EN=3
20 FOR I=1 TO EN
30 EN=10
40 PRINT I
50 NEXT

 1
 2
 3
```

## 3.6 GOSUB...RETURN

**Format:** **GOSUB** *lineNumber*
...
**RETURN**

**Description:** Branches to and returns from a subroutine at a particular line number.

**Limitation:** The maximum number of nested **GOSUB** statements is 256.

**Example:**

```
10 PRINT "HELLO"
20 GOSUB 40
30 END
40 PRINT "WORLD"
50 RETURN

HELLO
WORLD
```

## 3.7 GOTO

**Format:** **GOTO** *lineNumber*

**Description:** Branches to a line number.

**Example:**

```
10 PRINT "HELLO"
20 GOTO 40
30 PRINT "SAILOR"
40 PRINT "WORLD"

HELLO
WORLD
```

## 3.8 IF...THEN...ELSE

**Format:** **IF** *numExpression* **THEN** *{statements|lineNumber}* *[***ELSE** *{statements|lineNumber}]*
**IF** *numExpression* **GOTO** *lineNumber* *[***ELSE** *{statements|lineNumber}]*

**Description:** Executes statements depending on a condition. If the result of *numExpression* is not 0 (the result is rounded) then the *clause* after **THEN** or **GOTO** is executed, that is, the statements after **THEN** are executed or program execution branches to the line number after **THEN** or **GOTO**. If the result is 0 and **ELSE** was specified then the clause after **ELSE** is executed.

**Example:**

```
10 I=3
20 IF I>2 THEN 40
30 PRINT "HELLO"
40 PRINT "WORLD"

WORLD
```

**Example:**

```
10 I=3
20 IF I=2 THEN PRINT "TWO" ELSE PRINT "NOT TWO"
```

```
NOT TWO
```

IF-THEN statements may be nested.

**Example:**

```
10 X=1
20 Y=2
30 IF X>Y THEN PRINT "GREATER" ELSE IF X<Y THEN PRINT "LESS"
ELSE PRINT "EQUAL"
```

```
LESS
```

If the `IF` statement does not contain the same number of `THEN` and `ELSE` clauses, then each `ELSE` is matched with the closest `THEN`.

**Example:**

```
10 A=1
20 B=2
30 C=2
40 IF A=B THEN IF B=C THEN PRINT "A=C" ELSE PRINT "A<>C"
```

(prints nothing)

## 3.9 INPUT

**Format:** `INPUT` *[promptString{,|;}] variable[,variable]**

**Description:** Assigns input from the keyboard to one or more variables. When an `INPUT` statement is executed input is read from the keyboard until the RETURN key is pressed. Input for multiple variables is separated by a comma (`,`) character.

When *promptString* is specified followed by a semicolon (`;`) then *promptString* is printed followed by a question mark (`?`). When *promptString* is specified followed by a comma (`,`) then *promptString* is printed without a following question mark.

If the type of the input does not match the type of the specified variable then `?Redo from start` is printed and reading input from the keyboard is repeated.

**Example:**

```
10 INPUT "LENGTH OF EDGE";R
20 PRINT "AREA OF SQUARE:";R*R
30 GOTO 10
```

```
LENGTH OF EDGE? 4
AREA OF SQUARE: 16
LENGTH OF EDGE? HELLO
?Redo from start
LENGTH OF EDGE? 2
AREA OF SQUARE: 4
...
```

## 3.10 LET

**Format:** *[***LET** *]variable***=***expression*

**Description:** Assigns the result of an expression to a variable. The keyword **LET** is optional.

**Example:**

```
10 LET A=11
20 PRINT A
30 B=21
40 PRINT B

 11
 21
```

## 3.11 ON...GOSUB

**Format:** **ON** *numExpression* **GOSUB** *lineNumber[***,***lineNumber]**

**Description:** Branches to one of several line numbers containing subroutines. The line number to branch to is selected by the result of *numExpression*. If it is 1 (the result is rounded), then program execution branches to the first line number. If it is 0 or greater than the number of listed line numbers (but less than 256) then program execution continues at the statement after **ON**...**GOSUB**. If it is negative or equal or greater than 256 then an error occurs.

**Example:**

```
10 I=2
20 ON I GOSUB 40,50,60
30 END
40 PRINT "LONDON" : RETURN
50 PRINT "PARIS" : RETURN
60 PRINT "ROME" : RETURN

PARIS
```

## 3.12 ON...GOTO

**Format:** **ON** *numExpression* **GOTO** *lineNumber[***,***lineNumber]**

**Description:** Branches to one of several line numbers. The line number to branch to is selected by the result of *numExpression*. If it is 1 (the result is rounded), then program execution branches to the first line number. If it is 0 or greater than the number of listed line numbers (but less than 256) then program execution continues at the statement after **ON**...**GOTO**. If it is negative or equal or greater than 256 then an error occurs.

**Example:**

```
10 I=3
20 ON I GOTO 30,40,50
30 PRINT "LONDON" : GOTO 60
40 PRINT "PARIS" : GOTO 60
50 PRINT "ROME"
60 END

ROME
```

## 3.13 PRINT

**Format:** PRINT *[[expression]{;|,| }]**

**Description:** Prints the result of zero, one, or more expressions at the current cursor position. A semicolon (;) or a space character ( ) places the cursor immediately at the end of the previously printed *expression*. A comma (,) places the cursor at the beginning of the next *print zone* after the end of the previously printed *expression*. A print zone is a 14-character wide interval of cursor positions. If an *expression* does not end with a semicolon (;), space character ( ), or comma (,) the cursor is placed at the beginning of the next line of the printed *expression*.

Numbers are printed with a trailing space character. Positive numbers are printed with a leading space character.

**Example:**

```
10 PRINT "HELLO";" WORLD"
20 PRINT "HELLO";
30 PRINT " WORLD"
40 PRINT
50 PRINT "HELLO"," ","WORLD"
60 PRINT 123;"UNITS"
70 PRINT -123;"UNITS"
80 PRINT 1;2;3;4

HELLO WORLD
HELLO WORLD

HELLO           WORLD
 123 UNITS
-123 UNITS
 1  2  3  4
```

## 3.14 READ

**Format:** READ *variable[,variable]**

**Description:** Reads constants from a **DATA** statement and assigns them to variables. The constant type and variable type must match. If more constants are read than are present in **DATA** statements then an error occurs. To reread constants use the **RESTORE** statement.

**Example:**

```
10 FOR I=1 TO 3
20 READ A$
30 PRINT A$
40 NEXT I
50 DATA PARIS,ROME,LONDON

PARIS
ROME
LONDON
```

**See also:** DATA
RESTORE

## 3.15 REM

**Format:** `REM` *string*

**Description:** Insert a comment into the program.

**Example:**

```
10 REM *** CALCULATE THE AREA OF A SQUARE ***
20 EDGE=10
30 AREA=EDGE*EDGE
```

## 3.16 RESTORE

**Format:** `RESTORE` *[lineNumber]*

**Description:** Permits **READ** statements to reread constants from **DATA** statements. If *lineNumber* is specified then the next **READ** statement reads constants from the **DATA** statement at the specified line number on. If *lineNumber* is not specified then the next **READ** statement reads constants from the first **DATA** statement on.

**Example:**

```
10 FOR I=1 TO 3
20 READ A$
30 PRINT A$
40 NEXT I
50 RESTORE
60 FOR I=1 TO 3
70 READ A$
80 PRINT A$
90 NEXT I
100 DATA PARIS,ROME,LONDON

PARIS
ROME
LONDON
PARIS
ROME
LONDON
```

**See also:** DATA
READ

## 3.17 STOP

**Format:** `STOP`

**Description:** Stops the program; effectively the same as the **END** statment.

**Example:**

```
10 IF A=1 THEN STOP ELSE RETURN
```

## 3.18 SWAP

**Format:** `SWAP` *variable1*`,`*variable2*

**Description:** Exchanges the values of two variables. The variable types must match.

**Example:**

```
10 A=10
20 B=20
30 PRINT A;B
40 SWAP A,B
50 PRINT A;B

 10  20
 20  10
```

## 3.19 WHILE...WEND

**Format:**

`WHILE` *numExpression*

...

`WEND`

**Description:** Executes a sequence of statements repeatedly as long as a condition holds. If the result of *numExpression* is not 0 (the result is rounded) then the statements between **WHILE** and **WEND** are executed. When program execution reaches the **WEND** statement it branches back to the **WHILE** statement to check the result of *numExpression* again. If the result of *numExpression* is 0 then program execution continues at the statement after **WEND**. WHILE-WEND loops can be nested.

**Example:**

```
10 I=1
20 WHILE I<4
30 PRINT I
40 I=I+1
50 WEND

 1
 2
 3
```

# 4. Functions

This section lists all function of the implemented BASIC language.

## 4.1 ABS()

**Format:** `ABS(`*number*`)`

**Description:** Returns the absolute value of *number*.

**Example:**

```
PRINT ABS(3)
 3

PRINT ABS(-3)
 3
```

## 4.2 ASC()

**Format:** `ASC(`*string*`)`

**Description:** Returns the ASCII code of the first character of *string*. If *string* is an empty string (`""`) then an error occurs.

**Example:**

```
PRINT ASC("HELLO WORLD")
 72
```

## 4.3 ATN()

**Format:** `ATN(`*number*`)`

**Description:** Returns the arctangent of *number*. *number* is an angle in radians.

**Example:**

```
PRINT ATN(1)
 0.7853982
```

## 4.4 CHR$()

**Format:** `CHR$(`*number*`)`

**Description:** Returns a string whose single character is represented by ASCII code *number*. *number* is rounded and must be in the range of 0 to 127, otherwise an error occurs.

**Example:**

```
PRINT CHR$(65)
 A
```

## 4.5 COS()

**Format:** `COS(`*number*`)`

**Description:** Returns the cosine of *number*. *number* is an angle in radians.

**Example:**

```
PRINT COS(1)
 0.5403023
```

## 4.6 EXP()

**Format:** `EXP(`*number*`)`

**Description:** Returns *e* to the power of *number*. If *number* > 87.3365 then `Overflow` is printed, a value of positive infinity is returned, and execution continues.

**Example:**

```
PRINT EXP(1)
 2.718281
```

## 4.7 FIX()

**Format:** `FIX(`*number*`)`

**Description:** Returns the truncated integer part of *number*.

**Example:**

```
PRINT FIX(1.4)
 1

PRINT FIX(-1.4)
-1
```

**See also:** `INT()`

## 4.8 INSTR()

**Format:** `INSTR(`*[offset*`,`*]string*`,`*searchString*`)`

**Description:** Searches the first occurence of string *searchString* in *string* and returns the position at which the match starts. The first character of *string* has position 1. If no match was found then 0 is returned. The optional argument *offset* sets the start position of the search. *offset* must be in the range of 1 to 255, otherwise an error occurs. If *searchString* was not found, or *string* is empty, or *offset* is greater than the number of characters of *string* then 0 is returned. If *searchString* is empty then 1 or *offset* is returned.

**Example:**

```
PRINT INSTR("HELLO WORLD","L")
 3

PRINT INSTR(5, "HELLO WORLD","L")
 10
```

## 4.9 INT()

**Format:** `INT(`*number*`)`

**Description:** Returns the largest integer <= *number*.

**Example:**

```
PRINT INT(1.4)
 1

PRINT INT(-1.4)
-2
```

**See also:** `FIX()`

## 4.10 LEFT$()

**Format:** `LEFT$(`*string*`,`*length*`)`

**Description:** Returns a string composed of the *length* leftmost characters of *string*. *length* must be in the range of 0 to 255, otherwise an error occurs. If *length* is larger than the number of characters of *string* then the entire string *string* is returned. If *length* = 0 then an empty string (`""`) is returned.

**Example:**

```
PRINT LEFT$("HELLO WORLD",5)
HELLO
```

## 4.11 LEN()

**Format:** `LEN(`*string*`)`

**Description:** Returns the number of characters of *string*.

**Example:**

```
PRINT LEN("HELLO WORLD")
 11
```

## 4.12 LOG()

**Format:** `LOG(`*number*`)`

**Description:** Returns the natural logarithm of *number*. *number* must be > 0, else an error occurs.

**Example:**

```
PRINT LOG(2)
 .6931472
```

## 4.13 MID$()

**Format:** `MID$(`*string*`,`*offset[*`,`*length]*`)`

**Description:** Returns a string of *length* characters, beginning with the character at position *offset* of *string*. *offset* and *length* must be in the range of 1 to 255, otherwise an error occurs. If *offset* is greater than the number of characters of *string* then an empty string (`""`) is returned. If *length* is omitted or if there are fewer than *length* characters to the right of the character at position *offset* then all characters of *string* beginning with the character at position *offset* are returned.

**Example:**

```
PRINT MID$("HELLO WORLD",7,3)
WOR
```

## 4.14 POS()

**Format:** `POS(`*number*`)`

**Description:** Returns the current cursor position. The leftmost cursor position is 1. The argument *number* is ignored.

## 4.15 RIGHT$()

**Format:** `RIGHT$(`*string*`,`*length*`)`

**Description:** Returns a string composed of the *length* rightmost characters of *string*. *length* must be in the range of 0 to 255, otherwise an error occurs. If *length* is larger than the number of characters of *string* then the entire string *string* is returned. If *length* = 0 then an empty string (`""`) is returned.

**Example:**

```
PRINT RIGHT$("HELLO WORLD",5)
WORLD
```

## 4.16 RND()

**Format:** `RND(`*number*`)`

**Description:** Returns a random number between (including) 0 and (excluding) 1. If *number* > 0 then a new random number is returned. If *number* = 0 then the last random number is returned. If *number* < 0 then an error occurs.

**Example:**

```
PRINT RND(1)
 .9964446

PRINT RND(1) : PRINT RND(0)
 .6873739
 .6873739
```

## 4.17 SGN()

**Format:** `SGN(`*number*`)`

**Description:** Returns 1 if *number* > 0, 0 if *number* = 0, and -1 if *number* < 0.

**Example:**

```
PRINT SGN(2)
 1

PRINT SGN(0)
 0

PRINT SGN(-3)
-1
```

## 4.18 SIN()

**Format:** `SIN(`*number*`)`

**Description:** Returns the sine of *number*. *number* is an angle in radians.

**Example:**

```
PRINT SIN(1)
 0.841471
```

## 4.19 SPACE$()

**Format:** `SPACE$(`*number*`)`

**Description:** Returns a string composed of *number* space characters ( ). *number* is rounded to an integer value and must be in the range of 0 to 255, otherwise an error occurs.

**Example:**

```
A$=SPACE$(5) : PRINT "A";A$;"B"
A     B
```

## 4.20 SPC()

**Format:** `SPC(`*number*`)`

**Description:** Prints *number* space characters ( ). *number* is rounded to an integer value and must be in the range of 0 to 255, otherwise an error occurs. `SPC()` can be used only with the `PRINT` statement.

**Example:**

```
PRINT "A";SPC(2);"B"
A  B
```

## 4.21 SQR()

**Format:** `SQR(`*number*`)`

**Description:** Returns the square root of *number*. *number* must be >= 0, otherwise an error occurs.

**Example:**

```
PRINT SQR(2)
 1.414213
```

## 4.22 STR$()

**Format:** `STR$(`*number*`)`

**Description:** Returns a string that represents the value of *number*.

**Example:**

```
PRINT STR$(1.4)
 1.4

PRINT STR$(-1.4)
-1.4

PRINT "|";STR$(1.4);"|"
| 1.4 |

PRINT "|";STR$(-1.4);"|"
|-1.4 |
```

## 4.23 TAB()

**Format:** `TAB(`*number*`)`

**Description:** Advances the cursor to cursor position *number*. The leftmost cursor position is position 1. If the current cursor position is larger than position *number* then the cursor is placed in the next line before advancing to cursor position *number*. *number* must be in the range of 1 to 255, otherwise an error occurs. `TAB()` can be used only with the `PRINT` statement.

**Example:**

```
PRINT "HELLO";TAB(10);"WORLD
HELLO    WORLD

PRINT "HELLO";TAB(3);"WORLD
HELLO
  WORLD
```

## 4.24 TAN()

**Format:** `TAN(`*number*`)`

**Description:** Returns the tangent of *number*. *number* is an angle in radians. If `TAN()` results in a division by zero then `Division by zero` is printed, the value positive infinity or negative infinity is returned (depending on *number*), and execution continues.

**Example:**

```
PRINT TAN(1)
 1.5574077
```

## 4.25 VAL()

**Format:** `VAL(`*string*`)`

**Description:** Returns the numerical value of *string*. `VAL()` ignores leading whitespace characters. If *string* does not represent a number then `VAL()` returns 0.

**Example:**

```
PRINT VAL("1.4")
 1.4
```

•