BASIC

Beginner's All-purpose Symbolic Instruction Code

(This implementation)
by Trevin Beattie

Parts of this manual and much researched data comes from the book *Basic Computer Games* edited by David H. Ahl and copyright ©1978 by Creative Computing (ISBN: 0-89480-052-3). I have tried to avoid copyright infringement by generalizing as much as possible, but we must still give credit to this book as one of the best sources of the original language.

Table of Contents

1 Introduction

BASIC is one of the oldest high-level computer languages. The acronym B.A.S.I.C. stands for "Beginner's All-Purpose Symbolic Instruction Code"; it was designed to be easy to learn and use, and (in the author's opinion) is the best language for an aspiring programmer to start learning how to tell the computer to do things.

Since BASIC is an interpreted language, it is also handy for those 'quick-and-dirty' jobs a user has need for from time to time. Being *interpreted* means that when you type a statement in, it is executed right away. This is more convenient than *compiled* languages (such as 'C'), in which you must first write the whole program in a text editor, then compile the program, before you can run it.

Dartmouth is regarded as the original source of the BASIC programming language, written back in the late 50's or 60's for mainframe computers such as the DEC PDP-11 at educational institutions. When microcomputers hit the market, BASIC was ported to these computers as the primary programming language, and its popularity rose dramatically.

The only drawback to BASIC was that when it was ported to these dozen different microcomputers, the language was not standardized, so small variations were introduced from one machine to the next. (See ⟨undefined⟩ [Comprison with Other BASICs], page ⟨undefined⟩, for details.) In order to get a BASIC program from one computer to work on another, a user had to understand the way BASIC was implemented on both machines. Also, many microcomputer vendors added extensions to the BASIC language to take advantage of features available on their computers which were different from any other machine.

A couple of decades later (in the late 80's, I think) the American National Standards Institute (ANSI) attempted to address this problem by defining a standard for the BASIC language. Around this same time, however, some vendor had the bright idea of changing BASIC from an unstructured language to a structured one, which involved (among other things) getting rid of all line numbers. The resulting language looks *nothing* like any of the original microcomputer BASICs, and (to my knowledge) there is only one vendor which actually implements this standard: TrueBASIC.

In this implementation, I have attempted to return to the original implementations of BASIC as much as possible, so that program listings from the microcomputers of the 70's and early 80's can be entered and run on modern systems with little or no modification. The following chapters describe the BASIC language for the beginner, with notes about this specific implementation, and a comparison of other BASIC implementations.

2 General Syntax

2.1 Lines

A BASIC programs consists of lines of instructions. A *line* starts with a number, which determines the order in which the instructions are followed. For an example, we will introduce the BASIC statement `REM`, which lets you add a 'REMark' to the program. If you were to enter the following lines:

```
30 REM JACK FELL DOWN AND BROKE HIS CROWN
20 REM TO FETCH A PAIL OF WATER.
40 REM AND JILL CAME TUMBLING AFTER.
10 REM JACK AND JILL WENT UP THE HILL
```

then BASIC would interpret them in this order:

```
10 REM JACK AND JILL WENT UP THE HILL
20 REM TO FETCH A PAIL OF WATER.
30 REM JACK FELL DOWN AND BROKE HIS CROWN
40 REM AND JILL CAME TUMBLING AFTER.
```

By convention, when you are writing a program you would normally number your lines in multiples of ten. This will allow you or someone else to insert additional lines into the program at a later time.

Besides ordering, line numbers have one more very important function: many statements allow BASIC to 'jump' from one point in the program to another, skipping over intervening lines or even going back to a previous line in the program. These statements use line numbers to tell BASIC which line to execute next.

2.2 Statements

The simplest lines in BASIC consist of a single statement. Here are two lines—one which contains a `LET` statement, and the other contains a `PRINT` statement:

```
10 LET X=1+5/3
20 PRINT X
```

This program will first compute the value of X, and then print it (2.66667). A line can contain more than one statement; multiple statements are separated by a colon (':'). The previous example could also be written:

```
10 LET X=1+5/3:PRINT X
```

Because BASIC is an interpreted language, you don't have to prefix a statement with a line number. Any statement or group of statements you enter without a preceding line number is executed immediately by BASIC.

```
LET X=1+5/3:PRINT X
2.66667

READY
```

The `READY` line printed by BASIC indicates that BASIC has finished executing the instructions you have it and is ready to accept further instructions.

A Note On Case

As you have seen in the examples so far, BASIC programs are typically entered in all uppercase. Historically BASIC has been an uppercase-only language, because many of the earliest teletype machines did not have a lowercase character set.

Most of the old microcomputers had both upper- and lowercase characters, but by default all text was entered in uppercase. Some versions of BASIC were case-sensitive, meaning that if you attempted to enter a statement in lower case, the computer would not be able to understand you.

This implementation of BASIC is case-insensitive. You can enter statements in either uppercase or lowercase, or a mixture of both, and BASIC will be able to recognize them. When you list your program, all words that BASIC knows will automatically be converted to uppercase. The same applies to variable names (See Section 3.1 [Variables], page 5), except that variable names retain the same case the had when you first used them.

The exception to this is strings (See Section 2.3 [Data Types], page 3.) BASIC itself doesn't care what the contents of a string are, but when you compare two strings, lowercase letters are considered different than their uppercase counterparts.

To ensure compatibility with other BASICs, you should enter everything in upper case only.

2.3 Data Types

BASIC has only two data types: numbers and *strings* (any arbitrary sequence of numbers, digits, punctuation marks, etc.).

A simple number is entered in BASIC just as it would be in any other (computer or human) language. Numbers in BASIC can have a *floating point*, such

as `3.14159` (the value of π). You can also specify an exponential multiplier for exceptionally large or small numbers. The syntax for an exponential number is *num*`E`[`+`|`-`]*exp*, which means $num \times 10^{exp}$. So, for example, to enter the speed of light (c, 300 thousand kilometers per second), you could type *`3.0E+8`*.

Numbers in this implementation of BASIC can have any value up to 10^{308} in magnitude, and retain up to 15 digits of precision. (When BASIC prints out numbers, however, it only displays six significant digits. In program listings, floating-point numbers are also reduced to six significant digits, but integers less than 4 billion will show all digits.)

Strings are entered into programs by enclosing them in double quotation marks, '`"LIKE THIS"`'. A string can contain any regular character except another double quotation mark or `Return`.

3 Expressions

3.1 Variables

One of the nice features of programs is that the can repeat the same sequence of instructions using different starting values, thus generating different results. For example, to calculate the distance between a point at $(0, 0)$ and $(3, 4)$, you could enter the statement:

```
PRINT SQR(3*3+4*4)
5

READY
```

But what if you wanted to repeat the calculation with different end points, or even different starting points? To make a more general-purpose program, you would define a set of *variables* to represent the starting and ending points, and perform your computations on the variables instead of the actual numbers:

```
10 PRINT SQR((X1-X0)*(X1-X0)+(Y1-Y0)*(Y1-Y0))
```

You can see that this statement is very similar to the equations you studied in algebra. Now, when the program is supplied with any arbitrary starting and ending coordinates, it will show the distance between those points, and you don't have to change the formula.

BASIC has two basic variable types corresponding to its data types: numeric variables and string variables. A numeric variable name consists of a series of letters and numbers, with the rule that the first character in the name must be a letter. So, for example, a variable name can be as short as *X*, or as long as *A1ANDA2ANDA3ANDA4* or more! (However, we recommend keeping your variable names short.)

A string variable is named just like a numeric variable, but with a dollar sign ('$') tagged at the end, as in *A$* or *PLAYER2NAME$*. String variables are distinct from numeric variables, so, for example, you can use both *N* and *N$* in your program.

3.2 Arrays

At times you may wish to have a large set of related variables, for example elements in a vector. For this purpose, you can define a *variable array*, which is like a list of values. Unlike simple variables, which you can make up and use as you need them, an array must be defined ahead of time so that BASIC knows how big it is. This is done with the `DIM` (DIMension) statement:

```
10 DIM X(10),Y(10)
```

This example defines two arrays, X and Y, having ten elements each. To use a particular element of an array, just give the array 'subscript' in parentheses, as in '`X(1)`' or '`Y(4)`'. The subscript can even be another variable or an arithmetic expression, as in '`X(2*N-1)`'.

Variable arrays are distinct from simple variables, so you could, if you wanted to, use both X and *X(N)* in your program; the value of X is distinct from any of the elements of *X(N)*. For clarity, however, we do not recommend this.

An array can have more than one *dimension*. To define a multi-dimensional array, use the `DIM` statement just as with a one-dimensional array, and specify the size of each dimension separated by commas within the variable's parentheses. For example:

```
10 DIM M(7,9),N(5,5,5,5,5)
```

defines a two-dimensional matrix M with 7 rows and 9 columns, and a 5-dimensional array N where the size of each dimension is five (for a total of 5^5, or 3125 elements!). Remember that when you access an element of an array, you must provide the same number of subscripts as the number of dimensions in the array. If you give too few or too many subscripts, you will get an error.

You can re-dimension an array at any time, to give it a different size or even a different number of dimensions. Be warned, however, that when you do, all of the previous values contained in the array will be lost. A newly dimensioned numeric array will have all of its elements set to zero; a string array will start out with all of its strings empty.

3.2.1 Caveat - Subscript Starting Point

Older BASICs did not agree on how array elements should be numbered; some versions numbered the array subscripts from 1 to n (where n is the size of the array), while others numbered the subscript from 0 to $n - 1$. This implementation allows either origin; your arrays actually have elements numbered from 0 through n for a total of $n + 1$ elements. But you should restrict your indexing to either a 1-base or 0-base.

3.3 Assignment

The basic method of assigning a value to your variable is with the `LET` statement, as in the following example:

```
10 LET M(X(I),Y(I))=I
```

In this version of BASIC, like most microcomputer versions, the keyword `LET` is optional; so you can get the same result just by typing:

```
10 M(X(I),Y(I))=I
```

3.4 Operators

3.4.1 Mathematical Operators

BASIC understands the basic arithmetic operators '`+`' and '`-`' for addition and subtraction, and the characters '`*`' and '`/`' for multiplication and division. You can also use the '`^`' character for exponentiation; entering *`X^Y`* will yield the result of x^y.

Complex expressions are evaluated in standard algebraic form—operators of higher precedence ('`^`') are evaluated before operators of lower precedence ('`+`', '`-`'), and operators of equal precedence are evaluated from left to right.

The '`-`' operator serves a dual purpose. In addition to subtraction (no pun intended), it is also used to negate a number—to change a number from positive to negative or vice-versa. Negation, unlike all of the other operators, has a right-to-left association... but you wouldn't use multiple negative signs in a row now, would you?

You may use parentheses to change the order of evaluation. For example, the expression *`5+4/3*2`* is evaluated as $5 + ((4 \div 3) \times 2)$, resulting in 7.66667. To evaluate the formula $\frac{5+4}{3\times 2}$ you would type in *`(5+4)/(3*2)`*, which will give you 1.5.

3.4.2 Logical Operators

There are relational operators you can use to compare two numbers or expressions. These operators are normally used in the `IF..THEN` statement (See ⟨undefined⟩ [IF], page ⟨undefined⟩). Each of these operators results in a 'true' or 'false' answer, depending on whether the condition is true for the values on either side of the operator.

The relational operators are '`=`' (for equality), '`<>`' ($\neq$, for inequality), '`<`' (left value less than the right value), '`>`' (left value greater than the right value), '`<=`' ($\leq$, less than or equal), and '`>=`' ($\geq$, greater than or equal).

Note that in this implementation, in order to reduce errors due to rounding, comparisons are only done with six digits of precision, unless the magnitude of the number is greater than 10^{38} or smaller than 10^{-38}.

Relational operators also work with strings and string variables. In such comparisons, one string is considered 'less' than another if the first character which differs in the left string precedes the corresponding character of the right string according to their ASCII character codes. Since string comparisons are case-sensitive, all uppercase letters precede all lowercase letters; thus the expression '`"USA" < "United States"`' evaluates as true.

In addition to the relational operators, there are two boolean operators: `AND` and `OR`. These can be used with relational comparisons to test two conditions and evaluates to true if both or either condition is true.

The 'truth' result of a logical operation is actually a value of either zero or one. This is important when reading some existing BASIC programs; some variations of BASIC define 'true' to be −1, and some BASIC programs use this value directly in arithmetic expressions assuming one sign or the other. When you write a program, you should not assume that 'true' is negative or positive, or even 1 for that matter. The proper way to use a truth result in an expression is to first set the sign and magnitude with something like '`ABS(SGN(`*condition*`))`' (See Section 3.6.1.1 [ABS], page 9, and Section 3.6.1.4 [SGN], page 9).

3.4.3 String Operators

There is only one string operator defined in this version of BASIC, and it is present only because some popular BASIC programs use it. The '`+`' operator will *concatenate* two strings—it creates a new string consisting of the first and second strings run together. For example, if the contents of the variable *NAME$* is '`"JOHN"`', then the result of '`"HELLO "+NAME$`' is '`"HELLO JOHN"`'.

3.5 Precedence

For the record, here is the order of precedence of all the operators BASIC knows. Operators higher up in the list will be evaluated before those below them.

1. '`-`' (negation)
2. '`^`' (exponentiation)
3. '`*`', '`/`' (multiplication and division)
4. '`+`', '`-`' (addition and subtraction)
5. Relational operators*
6. '`AND`'
7. '`OR`'

*The relational operators have no associativity. Unlike the arithmetic and boolean logic operators, you cannot place two relational operators in the same subexpression. For example, the expression '`4<5=1<9`' simply makes no sense. Instead, you must explicitly parenthesize the subordinate relations, as in '`(4<5)=(1<9)`' (which is true). You do not need parentheses

for comparisons separated by '`AND`' and '`OR`'; it is okay to say '`4<5 AND 1<9`' (true).

3.6 Functions

Arithmetic will only get you so far. BASIC provides a set of *functions* which perform more complex calculations (such as trigonometry) and manipulation of strings.

3.6.1 Simple Numeric Functions

3.6.1.1 ABS

`ABS(X)` will compute the absolute value of X; if X is negative, the answer will be positive.

3.6.1.2 INT

`INT(X)` returns the integer part of X by truncating any digits past the decimal point (rounding towards zero).

3.6.1.3 RND, SEED

'`RND(1)`' generates a random number in the range $[0, 1)$. This corresponds with the implementation of most other BASICs. This version of `RND` has two additional variations: If you use '`RND(0)`', BASIC will first *randomize* the random number generator (otherwise, the sequence of random numbers would come up the same the next time you started BASIC).

Secondly, you can use '`RND(N)`' to generate a random number in the range $[0, N)$. This makes it a little easier to generate random numbers in an arbitrary range, rather than computing '`N*RND(1)`'.

If for some reason you need to repeat a sequence of random numbers (for example, to repeat a statistical analysis or to debug your program), you can use '`SEED(X)`' to *seed* the random number generator at a given starting point X.

3.6.1.4 SGN

`SGN(X)` will return one of three possible values: 1 if X is positive, -1 if X is negative, or 0 of X is zero.

3.6.2 Trigonometric and Transcendental Functions

In BASIC, all trigonometric functions measure angles in degrees (0–360).

3.6.2.1 ATN

`ATN(X)` computes the arctangent of X.

3.6.2.2 COS

`COS(`Θ`)` computes the cosine of Θ.

3.6.2.3 EXP

`EXP(X)` computes the natural exponential of X, or e^X.

3.6.2.4 LOG

`LOG(X)` computes the natural logarithm of X.

3.6.2.5 SIN

`SIN(`Θ`)` computes the sine of Θ.

3.6.2.6 TAN

`TAN(`Θ`)` computes the tangent of Θ.

3.6.3 String Functions

3.6.3.1 LEN

`LEN(A$)` returns the length of the string *A$*.

3.6.3.2 LEFT$

`LEFT$(A$,N)` makes a new string consisting of the first N characters of *A$*. If N is greater than the length of *A$*, you will get a copy of the entire string.

3.6.3.3 RIGHT$

`RIGHT$(A$,N)` makes a new string consisting of the *last* N characters of *A$*. If N is greater than the length of *A$*, you will get a copy of the entire string.

3.6.3.4 MID$

`MID$(A$,B,N)` makes a new string consisting of N characters from *A$*, starting with the B^{th} character. The first character in a string is numbered 1. If B is greater than the length of *A$*, you will get an empty string. If $B + N$ is greater than the length of *A$*, you will get the right part of *A$* starting at the B^{th} character.

3.6.4 Numeric and String Conversion Functions

3.6.4.1 ASC

`ASC(A$)` gives you the ASCII code of the first character in *A$*.

3.6.4.2 CHR$

`CHR$(X)` generates a 1-character string whose character's ASCII code is X. This is especially useful for printing a quotation mark ('`CHR$(34)`') or certain control characters (i.e., '`CHR$(7)`' for an ASCII bell).

3.6.4.3 STR$

`STR$(X)` makes the number X into a printable string.

3.6.4.4 VAL

`VAL(A$)` converts an ASCII representation of a number into a numerical value. Note that if A does not consist of (or at least start with) a number, the resulting value will be zero.

4 Statements

4.1 Remarks (REM)

REM adds a REMark to the program. BASIC ignores remarks when executing a program, but all text following the REM statement is preserved.

```
10 REM A GAME OF TIC-TAC-TOE BY JOHN DOE
```

Note that everything else up until the end of the line is considered part of the remark, so no other statement can follow REM. (You can use this to your advantage sometimes if you want to temporarily remove a line of statements from the program.)

Remarks help document what the program is doing. You should sprinkle these statements liberally throughout your program so that another person reading the code later on can understand it more easily.

4.2 Working With Data

4.2.1 LET

To assign a value to a variable (See Section 3.1 [Variables], page 5), use the LET statement. The syntax is '`LET` *`variable=expression`*'.

```
50 LET Y=5/9*(X-32)
```

In most versions of BASIC (including this one), the LET keyword is optional. If you omit it, BASIC will assume an *implied* LET. This can make your code simpler and in some cases easier to understand.

```
20 PI=3.14159265
60 F=C*9/5+32
```

4.2.2 DIM

Use DIM to create one or more arrays. The general syntax is '`DIM` *`variable`*`(`*`size`*`[,`*`size`*`...])[,...]`'. You may create any number of arrays on one line (at least as many as will fit). Arrays may be numeric or string. Each array must have at least one dimension; the maximum number of dimensions is limited by the amount of available memory. Each dimension can have a size as small as zero (not very useful) up to a maximum of 65535.

```
30 DIM X(10),R(2,3,5,7),A$(4,4)
```

4.2.3 DATA

The DATA statement defines a sequence of constant values that can be read sequentially by READ statements (described below). Each constant can be either numeric or string. Multiple constants can be listed on a single DATA line separated by commas.

DATA statements are ignored during program execution so they may be placed anywhere which is convenient or fits neatly with the code that uses it.

```
1000 DATA 42, 3.14159, "Hello"
```

4.2.4 READ

The READ statement assigns the next constant from a DATA statement to a variable. BASIC maintains an internal pointer to the datawhich starts at the first DATA statement and advances it to the next constant in the statement or the next line of DATA each time READ is used.

Multiple data values can be read at once by supplying several values to the READ statement separated by commas.

```
50 READ A, PI, GREETING$
```

If a READ statement attemps to read more data than is available, an error will occur. Likewise if the program tries to read a number into a string variable or a string into a numeric variable, an error will occur.

4.2.5 RESTORE

The RESTORE statement may be used to reset BASIC's internal data pointer back to the first DATA statement in a program. You can also provide a line number to this statement which will set the data pointer to the next DATA statement on or after the given line.

```
60 RESTORE
70 IF A < 42 THEN RESTORE 1000
```

Index

Index of Statements

Index of Functions