BASIC

Beginner's All-purpose Symbolic Instruction Code

(This implementation)
by Trevin Beattie

Table of Contents

1 Introduction 1

2 General Syntax 2

2.1 Lines 2
2.2 Statements 2
A Note On Case 3
2.3 Running Your Program 3
2.4 Data Types 4
2.5 Exiting Out of BASIC 4

3 Expressions 5

3.1 Variables 5
3.2 Arrays 6
3.2.1 Caveat - Subscript Starting Point 6
3.3 Assignment 7
3.4 Operators 7
3.4.1 Mathematical Operators 7
3.4.2 Logical Operators 7
3.4.3 String Operators 8
3.5 Precedence 8
3.6 Functions 9
3.6.1 Simple Numeric Functions 9
3.6.1.1 ABS 9
3.6.1.2 INT 9
3.6.1.3 RND, SEED 9
3.6.1.4 SGN 9
3.6.2 Trigonometric and Transcendental Functions 10
3.6.2.1 ATN 10
3.6.2.2 COS 10
3.6.2.3 DEGTORAD 10
3.6.2.4 EXP 10
3.6.2.5 LOG 10
3.6.2.6 RADTODEG 10
3.6.2.7 SIN 10
3.6.2.8 SQR 10
3.6.2.9 TAN 10
3.6.3 String Functions 10
3.6.3.1 LEN 10
3.6.3.2 LEFT$ 10
3.6.3.3 LOWER$ 11
3.6.3.4 RIGHT$ 11

3.6.3.5 MID$ 11
3.6.3.6 TAB 11
3.6.3.7 UPPER$ 11
3.6.4 Numeric and String Conversion Functions 11
3.6.4.1 ASC 11
3.6.4.2 CHR$ 11
3.6.4.3 STR$ 12
3.6.4.4 VAL 12
3.6.5 User-Defined Functions 12

4 Statements 13

4.1 Remarks (REM) 13
4.2 Working With Data 13
4.2.1 LET 13
4.2.2 DEF 13
4.2.3 DIM 14
4.2.4 DATA 14
4.2.5 READ 14
4.2.6 RESTORE 14
4.3 Input and Output 15
4.3.1 PRINT 15
4.3.2 INPUT 16
4.4 Flow Control 17
4.4.1 GOTO 17
4.4.2 END 18
4.4.3 Subroutines: GOSUB and RETURN 18
4.4.4 Temporary Pause: STOP and CONTINUE 19
4.4.5 Conditional Code: IF 19
4.4.6 ELSE 20
4.4.7 Looping with FOR and NEXT 20
4.4.8 Case Switching with ON / GO... 22
4.4.9 Timed delay 23
4.5 Program Management 23
4.5.1 Listing Your Program 23
4.5.2 Saving Your Program 23

Index 25

Index of Statements 27

Index of Functions 28

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⟩ [Comparison 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 (See Section 4.4.1 [GOTO], page 17.) 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 they had when you first used them.

The exception to this is strings (See Section 2.4 [Data Types], page 4.) 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 Running Your Program

After entering a complete program using line numbers, you tell BASIC to run the program by entering "`RUN`" after the `READY` prompt. Basic will start executing statements from the first line of the program.

Before executing the first statement, this version of BASIC clears all variables that the program uses as well as removing any user-defined functions.

`RUN` may also be used within a program, which will cause BASIC to start it over from the beginning.

2.4 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. Strings *can* contain `Return` (newline) characters; some older programs relied on this capability to print multiple lines with a single statement. For readability and portability, you should not rely on this behavior; use a separate `PRINT` statement for each line instead.

2.5 Exiting Out of BASIC

When you are finishing using the BASIC programming language, enter “`BYE`” at the `READY` prompt to leave the interpreter.

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.

Whenever you start a program with the `RUN` command, BASIC will clear all variables—numeric variables are set to 0 and string variables are emptied. However, not all versions of BASIC behave this way; programs should always initialize the variables they use at the beginning and not rely on them having any particular value.

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 should be defined ahead of time so that BASIC knows how big it is. This is done with the `DIM` (DIMension) statement:

```
10 DIM X(20),Y(20)
```

This example defines two arrays, *X* and *Y*, having twenty 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)`'.

If you use an array variable without first giving its `DIM`ension, BASIC will create it with a default dimension of 10 elements.

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 Section 4.4.5 [IF], page 19). 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 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 this implementation of BASIC, all trigonometric functions measure angles in radians (0–6.283185). Be aware that some dialects measured in angles in degrees, so programs written for those will need adjustments; the `DEGTORAD(D)` and `RADTODEG(R)` functions are provided to help with thsee conversions.

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 DEGTORAD

`DEGTORAD(A)` converts an angle A in degrees to an angle in radians.

3.6.2.4 EXP

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

3.6.2.5 LOG

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

3.6.2.6 RADTODEG

`RADTODEG(R)` converts an angle R in radians to an angle in degrees.

3.6.2.7 SIN

`SIN(Θ)` computes the sine of Θ.

3.6.2.8 SQR

`SQR(X)` computes the square root of X, or $\sqrt{X}$.

3.6.2.9 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 `LOWER$`

`LOWER$(`*A$*`)` makes a new string that is a copy of *A$* with all upper case characters converted to lower case. This function is not a part of the original BASICs, but was added because typing in lower case is much more common on modern computers.

3.6.3.4 `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.

3.6.3.5 `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.

3.6.3.6 `TAB`

`TAB` is a special function that is only available in `PRINT` statements (Section 4.3.1 [PRINT], page 15). It advances the cursor to the given column on the screen; for example `PRINT TAB(36);"CENTERED"` will print 36 spaces (assuming it starts from the left side of the screen) before the word "CENTERED".

If you end a `PRINT` command with `TAB`, it will remain at the given column rather than advance to the next line the same as if you had ended it with a semicolon (`;`).

3.6.3.7 `UPPER$`

`UPPER$(`*A$*`)` makes a new string that is a copy of *A$* with all lower case characters converted to upper case. This function is not a part of the original BASICs, but was added because typing in lower case is much more common on modern computers, but older BASIC programs typically expect input to be in upper case and string comparisons are case-sensitive.

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.

3.6.5 User-Defined Functions

A program can define its own functions by using the `DEF` statement. This looks similar to an assignment with an array variable, except the arguments are function parameters rather than an array index.

For example, the following program defines functions that select the minimum and maximum between two numbers:

```
10 DEF MIN(A,B)=(A+B-ABS(A-B))/2
20 DEF MAX(A,B)=(A+B+ABS(A-B))/2
```

The variables used for the function parameters are not important outside of the definition; for example if the variables `A` and `B` are used elsewhere with the above program their values will not be changed by calls to `MIN(X,Y)` or `MAX(I,J)`. The function definitions *may* use other variables besides the function parameters, in which case it will use the value that those variables hold at the time the function is called.

You can also define string functions this way. For example, the following program defines a function that generates a random letter from “A” to “Z”:

```
30 DEF RNDALPHA$(X)=CHR$(INT(RND(26))+65)
```

As shown in this example the function parameter doesn’t need to actually be used in the function, but the function definition must have at least one parameter.

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.

Some early versions of BASIC supported the longer keyword "REMARK" as well as its 3-letter abbreviation, so this version of BASIC supports this as well as "REM." for entering but will shorten it to the standard REM keyword in the program listing.

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 DEF

Use DEF to declare a user-defined function; see Section 3.6.5 [User-Defined Functions], page 12, for more details.

4.2.3 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)
```

4.2.4 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"
```

4.2.5 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.

4.2.6 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
```

4.3 Input and Output

4.3.1 PRINT

The `PRINT` statement is how a BASIC program informs a user about what it's doing or the results of the program. In its simplest form, `PRINT` with no data just outputs a blank line. To print a literal line of text, follow `PRINT` with a quoted string containing the text you want the program to print. This is the most common beginner program:

```
10 PRINT "Hello, world!"
```

A more useful case is to have the program print the value of a variable that has been calculated. Just follow `PRINT` with the name of the variable to display.

```
20 LET ROOT2=SQR(2)
30 PRINT ROOT2
```

To make it clearer what data is being shown, you can combine literal strings with variables. Separate the strings and variables with a semicolon to have them printed adjacent to each other.

```
30 PRINT "The square root of 2 is "; ROOT2
```

In some early versions of BASIC*, using the semicolon to separate strings and variables in* `PRINT` *was optional. This* BASIC *supports that syntax, but we encourage using the semicolon for readability.*

If you end the `PRINT` statement with a semicolon, BASIC will not start a new line at the end; another `PRINT` statement later in the program will continue printing on that same line. Remember to add a space at the end of a string if it will be followed by a variable value, otherwise they will be mashed together like "...is1.41421" instead of "...is 1.41421".

Likewise if you end `PRINT` with a `TAB` (see Section 3.6.3.6 [TAB], page 11), BASIC will leave the cursor at the given column instead of going to the next line so the next `PRINT` will start from that position.

You can space items out in the output at tab stops by separating them with a comma instead of a semicolon. The comma causes the output to advance to the next tab stop, which by convention is at the next multiple of 8 spaces from the start of a line.

```
PRINT "The first prime numbers are", 2, 3, 5, 7
```

The output from the above statement will look like:

```
The first prime numbers are     2       3       5       7
```

You may also begin the PRINT statement with a comma to advance to the next tab stop before printing the next item, or use several commas in a row to advance multiple tab stops.

```
10 PRINT "Flush left"
12 PRINT ,"Tab"
14 PRINT ,,"Tab"
16 PRINT ,,,"Tab"
18 PRINT ,,,,"Tab"
```

The output from running the above program will be:

```
Flush left
        Tab
                Tab
                        Tab
                                Tab
```

Ending a PRINT statement with a comma will advance to the next tab stop and, like the semicolon, leave the cursor there for the next PRINT instead of advancing to the next line.

4.3.2 INPUT

The INPUT statement lets the program get data or text from the user. This statement must be followed by the name of a variable in which to store the user's data. It can either get numeric or string data.

```
40 INPUT X
```

Basic will display a "? " prompt to let the user know it's waiting for input. To make it clear to the user what the program is asking for, you can either precede the INPUT statement with a PRINT statement (ending the PRINT with a semicolon if you want the "? " and user input to appear on the same line) or for convenience you can lead the `INPUT` statement itself with a literal string and a semicolon before the variable to use.

```
40 INPUT "How many apples do you have"; X
```

`INPUT` can read multiple variables in the same statement. Separate the variables with a comma. For each variable, BASIC will prompt the user with a “? ” and read the next line of input.

```
50 PRINT "Please enter your first, middle, and last name"
60 INPUT FIRST$, MIDDLE$, LAST$
70 REM Build the message incrementally
80 LET MESSAGE$="Hello " + FIRST$
90 MESSAGE$=MESSAGE$ + ", also called " + MIDDLE$
100 MESSAGE$=MESSAGE$ + ", of the " + LAST$ + " family!"
110 PRINT MESSAGE$
```

4.4 Flow Control

4.4.1 `GOTO`

Normally BASIC executes each line of a program in sequential order, and after running the last line of the program, it stops. This order can be changed by using the `GOTO` statement to tell BASIC which line of the program to execute next. A typical case is to have the program repeat the same code continually.

```
10 LET COUNT=1
20 PRINT COUNT,"Hello, world!"
30 COUNT=COUNT+1
40 GOTO 20
```

A program like the above will run forever unless the user interrupts it by typing Ctrl-*C*. Another use for `GOTO` is to skip over a section of code.

```
10 PRINT "One upon a time,"
15 GOTO 990
20 PRINT "there was a beautiful princess."
30 INPUT "What is your name"; NAME$
40 PRINT "Amazingly, the princess was also named "; NAME$
50 PRINT "In the kingdom of ..."
REM more story lines here
990 PRINT "And they lived happily ever after."
```

The program above will skip from the opening phrase right to the end without doing anything in the middle.

GOTO statements should only be used sparingly, since if a program is large and the flow is complex they can create what is known as “spaghetti code” where a developer can’t follow what the program is supposed to be doing, and in many cases in can break the flow of other control statements.

4.4.2 END

The END statement simply tells BASIC to stop running the program.

```
10 INPUT "What is your name"; NAME$
17 END
20 PRINT "Hello, "; NAME$
```

The above example will take the user’s input but never says “Hello”.

4.4.3 Subroutines: GOSUB and RETURN

BASIC can switch to another section of a program on a temporary basis and remember where it left so that it can come back and continue from there. The GOSUB statement works similarly to the GOTO statement, except it leaves a bookmark of where it left so that it can RETURN there later. This is very useful when you have a section of code that can be re-used in several places within your program.

The subroutine should end with a RETURN statement, which tells BASIC to go back to the next statement after the GOTO which sent it there.

```
10 INPUT "Enter the first value"; A
20 GOSUB 100
30 INPUT "Enter the second value"; A
40 GOSUB 100
50 INPUT "Enter the third value"; A
60 GOSUB 100
90 END
100 REM This subroutine computes functions of A
110 B=A*A
120 C=A*B
130 D=B*C
140 PRINT A; " squared is "; B
150 PRINT A; " cubed is "; C
160 PRINT "The fifth power of "; A; " is "; D
170 RETURN
```

4.4.4 Temporary Pause: STOP and CONTINUE

The STOP statement *temporarily* stops running a program, returning to the "READY" prompt. BASIC remembers where it left off.

After a program has stopped in this way, the CONTINUE statement resumes running the program from the next statement. If the program was finished (either by reaching the end of the program or using the END statement), CONTINUE does nothing.

For convenience, the CONTINUE command may be abbreviated as "CONT.".

4.4.5 Conditional Code: IF

Frequently a program will need to choose to do different things depending on the values of its variables. The IF statement evaluates an expression — typically a comparison — and either executes the statement following THEN if the result is true (i.e. has a non-zero value) or skips that statement if the result is false (i.e. zero). There are two forms of this statement.

```
120 IF X > 10 THEN 150
130 PRINT X;" is less than 10"
140 GOTO 160
150 PRINT X;" is greater than or equal to 10"
160 REM Both branches join here
```

When THEN is followed by a line number, the program goes to that line if the expression is true; otherwise it continues to the next statement. This is an implied GOTO.

In some old programs, you may find an implied GOTO followed by another statement on the same line, e.g. "IF S>4 THEN 770:IF S<1 THEN 770". This implementation of BASIC treats it as if there were an implied ELSE (see Section 4.4.6 [ELSE], page 20) instead of the colon (":"), but programs should not rely on this behavior and write statements a different way for clarity. The preceding statement would be better written "IF S>4 OR S<1 THEN 770".

```
200 IF Y < 10 THEN PRINT Y;" is less than 10"
210 IF Y > 10 THEN PRINT Y;" is greater than 10"
```

When THEN is followed by another statement, the program executes the rest of the line if the expression is true; otherwise it skips that line and continues to the next line. You may have multiple statements after THEN; BASIC will skip all of them if the expression is false.

The statement following THEN may be another IF statement, allowing you to chain multiple conditions together. This is for compatibility with some older Dartmouth BASIC programs; for clarity in new programs, consider using the `AND` operator instead.

For compatibility with some versions of BASIC, then `THEN` keyword may be omitted; for example:

```
250 IF A<1 GOTO 520
```

However in most cases leaving out "`THEN`" can render the code less readable, so it is recommended that you always use it.

4.4.6 `ELSE`

`ELSE` is used following an `IF` / `THEN` statement to have BASIC run an alternative statement if the expression is false (zero). As with `THEN`, there are two forms of this clause available.

```
300 IF B < C THEN 320 ELSE 350
```

When `ELSE` is followed by a line number, the program goes to that line if the `IF` expression is false. This is an implied `GOTO`.

```
10 IF A < 0 THEN END ELSE PRINT "Still going..."
```

When `ELSE` is followed by another statement, the program executes the rest of the line if the `IF` expression is false.

Note that the grammar only allows one statement (or line number) between `THEN` and `ELSE`, with no colons. If you need to have multiple statements in the condition true case, break up the code blocks over multiple lines.

The statement following `ELSE` may be another `IF` statement, allowing you to chain multiple conditions together. However, nesting `IF` statements this way can make your program confusing to read since it may not be clear which condition a secondary or later `ELSE` clause applies to. Consider using a single conditional expression with `AND` and `OR` operators instead, or using implied `GOTO`s and separating the true and false statement blocks over different lines.

4.4.7 Looping with `FOR` and `NEXT`

A section of code can be repeated a specified number of times by enclosing it in a `FOR`. . .`NEXT` loop. The `FOR` statement tells BASIC what variable to use for counting the number of times the loop runs; it sets the given variable to the initial value and then continues with the following statements until reaching the `NEXT` statement for the same variable. Then it adds 1 to the variable's value and checks whether it has passed the `TO` value; if not, BASIC goes back to the statement following `FOR`.

```
10 FOR A=1 TO 5
20 READ STR$
30 PRINT A;". ";STR$
40 NEXT A
50 DATA "Pink hearts","Yellow moons"
51 DATA "Orange stars","Green clovers"
52 DATA "Blue diamonds"
```

The above example will print 5 numbered lines, each showing the next string from the data.

You can change the amount and/or direction by which the variable increments each time through the loop by adding a `STEP` to the `FOR` statement. Then BASIC will add the `STEP` amount to the variable each time it reaches `NEXT`. For example, the following code will count backwards:

```
10 FOR N=100 TO 1 STEP -1
20 PRINT N;
30 IF N>1 THEN PRINT ", ";
40 T=N/10:T=T-INT(T)
50 IF T=0.1 THEN PRINT
60 NEXT N
```

And the next example will show multiples of π:

```
10 FOR P=0 TO 32 STEP 3.14159
20 PRINT P
30 NEXT P
```

`FOR` loops can be nested if you use different variables.

```
10 DIM M(10,10)
20 FOR I=1 TO 10
30 FOR J=1 TO 10
40 IF I=J THEN M(I,J)=1:ELSE M(I,J)=0
50 NEXT J
60 NEXT I
```

Some older variants of BASIC allowed you to list multiple variables in the same `NEXT` statement. For compatibility this BASIC also allows it, so for example

```
50 NEXT J,I
```

is equivalent to the two separate NEXT statements in the previous example. Some variants also allowed a NEXT statement *without* specifying a variable; this would repeat the most recent (innermost) NEXT loop. This implementation of BASIC supports the bare NEXT syntax as well, but it renders the code less readable and is discouraged.

In the event that the initial value of a FOR loop exceeds the TO value (with a positive STEP, or if it is less than the TO value with a negative STEP) BASIC will skip over the following statements until it finds the matching NEXT statement for the same variable, then continue the program after that point.

```
10 LET S=-1
20 FOR X=0 TO S
30 PRINT "This line should not be shown"
40 NEXT X
50 PRINT "This should be the first line shown"
```

4.4.8 Case Switching with ON / GO...

The ON statement lets a program choose one of many possible paths to take next depending on a numeric expression. ON is followed by a variable or expression that evaluates to an ordinal number that tells BASIC which entry in a list of line numbers to use, then either GOTO (see Section 4.4.1 [GOTO], page 17) or GOSUB (see Section 4.4.3 [GOSUB and RETURN], page 18) followed by the list of line numbers.

```
10 FOR X=1 TO 5
20 ON X GOSUB 1100,1200,1300,1400,1500
30 NEXT X
40 END
```

In the above example, the program will go to each of the subroutines in the list as X runs through the sequence of numbers 1 through 5.

Any fractional value for the expression is truncated to an integer. If the expression evaluates to a non-positive number, BASIC will go to the first line number in the list. If the expression is greater than the number of line numbers in the list, BASIC will skip over the GOTO / GOSUB part of the ON statement and continue to the next statement.

4.4.9 Timed delay

On modern computers, BASIC runs many orders of magnitude faster than it did on old 8-bit computers from the 1970's or mainframes of the 1960's. Some programs relied on busy do-nothing loops to add delays in programs that had repeating events like animation. In this implementation of BASIC, we've provided a `PAUSE` statement which will cause BASIC to sleep for a specified number of seconds, which are specified as a decimal number. For example,

```
100 PAUSE 12.345
```

will have BASIC sleep for 12 seconds and 345 milliseconds. The user may interrupt this pause by pressing `Ctrl-C`, which will also stop the program.

4.5 Program Management

4.5.1 Listing Your Program

The `LIST` statement displays the contents of your BASIC program. Without any parameters, `LIST` will show the entire program.

If your program is large, you may want to show just part of the program at a time. With a single numeric argument, `LIST` shows the specified line. With two numbers, e.g. `LIST 100,199`, it shows all lines in that range.

4.5.2 Saving Your Program

To save your program for later use, use the `SAVE` command with a string giving the name of a file in which to store the program. BASIC will produce a listing of the program like the `LIST` command, but write it to the named file instead of the screen.

```
SAVE "MyPrograms/example.basic"
```

How you name your program files is entirely up to you, but using an extension of `.basic` or `.BASIC` is recommended to identify them as BASIC programs. If you don't use `/` at the beginning of the name the file is relative to the directory where you started `basic` from.

To clear the current program from memory, use the `NEW` command.

You can also erase individual lines from the program by entering the line number without any statements on it.

To load a previously saved program back into memory, use the `LOAD` command with a string giving the name of the file containing the program listing. BASIC will read this file just as if it were being typed into the console. This means the file may even contain unnumbered statements for immediate execution; for example a program file may end with the line "RUN" to run the program immediately after it finishes loading.

```
LOAD "MyPrograms/example.basic"
```

If you already have a program in memory, you may want to clear it first with NEW before loading the new program; otherwise the new code being loaded will be merged with any existing code. This can be useful for example if you have modifications to a program in a separate file from the original code. The sample program "Animal" is a good example of this; the game can be extended by adding more DATA lines with additional questions and answers.

```
NEW
LOAD "examples/Animal.BASIC"
LOAD "examples/Animal.more"
```

Index

A

ANSI 1
arithmetic operators 7
array subscript base 6
arrays 6, 14
assignment of values to variables 13

B

B.A.S.I.C., definition of 1
boolean operators 8

C

case sensitivity 3
comments in the program 13
conditional execution 19

D

Dartmouth BASIC 1
delay 23

E

exiting BASIC 4
exponent 4
exponentiation 7

F

floating-point 4
flow control 17
functions, conversion 11
functions, numeric 9
functions, string 10
functions, transcendental 10
functions, trigonometric 10
functions, user-defined 12

I

implied `GOTO` 19, 20
implied `LET` 13
input 16
interpreted 1, 3

J

jump to another line 17
jump with return 18

L

loops 20

N

negation 7
number 4

O

operator precedence 8
output 15

P

precedence, operator 8
predefined data lists 14
printing messages 15

Q

quitting BASIC 4

R

reading user data 16
`READY` 3
relational operators 7
remarks 13

S

selective jump 22
speed of light 4
standardization 1
stopping 18, 19
storing programs 23
string 4
string concatenation 8
subroutines 18
switching 22

T

timed delay 23
TrueBASIC 1

U

unconditional branch 17

V

variable arrays 6
variables 5
variables, assigning values to 13

Index of Statements

B

BYE 4

C

CONTINUE 19

D

DATA 14
DEF 12, 13
DIM 14

E

ELSE 20
END 18

F

FOR 20

G

GOSUB 18
GOTO 17

I

IF 19
INPUT 16

L

LET 2, 7, 13
LIST 23
LOAD 23

N

NEW 23
NEXT 20

O

ON...GOSUB 22
ON...GOTO 22

P

PAUSE 23
PRINT 2, 15

R

READ 14
REM 2, 13
RESTORE 14
RETURN 18
RUN 3

S

SAVE 23
STEP 20
STOP 19

Index of Functions

*

* . . . 7

+

+ (for addition) . . . 7
+ (for string concatenation) . . . 8

–

- (for negation) . . . 7
- (for subtraction) . . . 7

/

/ . . . 7

<

< . . . 7
<= . . . 7
<> . . . 7

=

= . . . 7

>

> . . . 7
>= . . . 7

^

^ . . . 7

A

ABS . . . 9
AND . . . 8
ASC . . . 11
ATN . . . 10

C

CHR$ . . . 11
COS . . . 10

D

DEGTORAD . . . 10

E

EXP . . . 10

I

INT . . . 9

L

LEFT$ . . . 10
LEN . . . 10
LOG . . . 10
LOWER$ . . . 11

M

MID$ . . . 11

O

OR . . . 8

R

RADTODEG . . . 10
RIGHT$ . . . 11
RND . . . 9

S

SEED . . . 9
SGN . . . 9
SIN . . . 10
SQR . . . 10
STR$ . . . 12

T

TAB . . . 11
TAN . . . 10

U

UPPER$ . . . 11

V

`VAL` 12