Jim Hawkins Dartmouth BASIC Language Interpreter V 1.1 2021
Updated 12/31/2021

PREFACE

This interpreter is strongly based, in style, to the original Dartmouth BASIC, designed by two professors at Dartmouth College, John G. Kemeny, released in 1964 at Dartmouth College and is intended as a quick and handy interpretive (write and run) language. It is extended from the original BASIC, but not so far extended as to be unrecognizable as the original BASIC as many other Good products given the BASIC name, such as Visual BASIC. The BASIC name is an acronym that stands for **Beginners' All-purpose Symbolic Instruction Code**. It consists of simple, line numbered statement or command strings of the form:

Line Number	Statement	Expression………………………..

Line Number Statement Expression or, in particular:

100 if a > b goto 400

It is not intended to be a language for game development, although it wouldn't be impossible. It is also not a good starting language for those seeking to build a career in programming in well structured code as with C, C++, Java, or Python. It is written to provide the user with an easy program language that can be handy for writing one's own utilities. The first example at the end is a Celsius to Fahrenheit conversion table generator. It both displays the table on the monitor and writes the table into a file, which can be printed.

Presently, it is implemented as a Windows console application, written in the C programming Language. It has never been marketed to the public. It was the first, high level programming language I wrote programs for in 1975, which started my 30 year software development career in many languages. I intend to "Windowize" it with a friendly GUI. I also intend to port it to run on the Linux operating system as a command line executable and add some more powerful commands, such as matrix manipulation or MAT commands. Although it presently does not offer a precision graphing capability, it is possible to create crude character based graphs consisting of ASCII illustrated in two figures below.

```
          *
              *
                *
               *
            *
       *
    *
    *
     *
        *
```

OR:

```
================|
======================|
=========================|
========================|
===================|
============|
=======|
======|
=========|
================|
======================|
=========================|
========================|
===================|
============|
=======|
```

It can serve as a quick algebraic expression computer as in:

print 3.14*33/2*sin(30)*(33+22+11)

and so on. Notice that it can be typed in without a line number for immediate calculation. I wrote the expression evaluator. The precedence of operators are guided by a simple operator precedence table.

I wrote the original version to work on my DOS computer, many years ago, but the saved source code just sat there as I moved into Windows. It has been a challenging job just to port it as a console application written in the C programming language as C has evolved to meet an ANSI standard with more strict rules. In the process of testing it, I found many “bugs” and fatal errors causing program crashes. Most of them are fixed, but I know there are more. Some functions were completely rewritten in the last four months.

I plan to offer it as open source, for others to play with or expand, but not until I post some algorithms for others to use, free of any license or charge.

BEFORE GETTING STARTED

Note that this program is a console program that runs only on machines with Windows installed. I plan to make it a Windows app and compile it to run on Linux. It will take some time because, the gcc Linux compiler has many different code requirements and standards. It isn't an instant port like it used to be when compilers were not as finicky as they are now.

File paths **use backslashes and disk drives, to denote file paths such as**

\ folder1\folder2\file

OR

c:\folder\file

Desktop icon **It makes it easier to create a desktop icon by right clicking on the BASIC executable and choosing Send to > desktop icon and/or pin to Start menu**

Improving console appearance **When you first start this program, you will undoubtedly get a tiny black console window. Click on the upper left corner icon, select and chose "size" as 24 or 28. You can even chose Font > Bold font. After making these settings, they will be "remembered" and the app will start with these settings.**

1. Conventions

This Manual: All things enclosed in [ ] are optional

expr Any algebraic expression which could be a constant, variable, math function or a combination of the aforesaid, separated by arithmetic operators as in:

a+b*3.14*(4.4+c2*sin(b+s)) +a(2,2) See "variables' and "math functions" below.

Note that whenever an array is specified with dimension(s) containing variables, those variables must have been assigned a value through *let* or *read* commands.

Operators: + - * / % or ^ for addition, subtraction, multiplication, division, modulo, or exponentiation in. order of lowest to highest precedence. + and - have the sameprecedence and *, /, and % have the same precedence. Parenthesis () around expressions forces the contents to be higher precedence than all parts of the expression outside, those parenthesis. Note also that when the '-' is used as a unary it maintains its low precedence, hence the expression -2^2 yields 4 instead of 4. In all cases a good rule of thumb to ensure precedence is to enclose the part of high precedence in parenthesis, thereby (-2)^2 yields 4. ,

Relationals: <, >, =, <=, > =, <>, *or*, *and*, *||*, *&&* for less than, greater than, equality, less or equal, .greater than or equal, not equal, logical *or* and logical *and*. Source Program Name The source program name is suffixed by a **.bas**

Statement A basic statement consists of a line number. (integer value between 1 and 65534 (on 32 bit machines) followed by a command; space and operand which follows the syntax governed by the command as in:

```
100 print "Hello World"
```

A statement can be typed without a line number in which case it will execute immediately. This is true for all commands, but doesn't make sense for a command such as *fo*r. Immediate execution is handy for diagnostic purposes such as *print* x. for some commands.

Strings Sequences of ASCII characters, enclosed by double quote characters.

Variables All variable names are either a lower case alpha character (a-z) or a lower-case alpha character followed by an integer (0-9). Arrays have the same name convention as regular variables and take the form: varname (expr1, expr2, expr3. . . .expr 10) where expr1 - expr 10 are the dimension attributes of the array and can take the form of any legal expression. Array variables are first allocated with the dim command.

2. Commands

2.1 Standard Commands

auto [start line#] [,increment] Turns on auto line number and increment] Defaults to 10,10
Terminate with a decimal point or **period '.'**

bye or **q** Exit the interpreter. The 'q' command will give you the chance to confirm that you want to quit. Bye will just quit.

com [mon] Preserve variables for subsequent run. Issue of the run command otherwise de-allocates all variables. NOT IMPLEMENTED

con [line#] Continue normal execution from single step mode. See *step* command.

data (expr), (expr), (expr) ....

The *data* statement is a string ,of defined constants or expressions referred to by the "read" statement. Unlike most BASIC interpreters, the data is stored only in the form of text strings which allows the read statement to evaluate expressions as well as constants.

del [ete] lownum [,highnum]

Delete line-number specified if only lownum given. Delete all lines between lownum and highnum if both are specified. See the undo command. If the lownum doesn't exist, it starts the deletion of the next higher line number.

dim variable (expr1 ,expr2........,expr10)

Allocate space and define the dimensional characteristics of subscripted variable. Keep in mind that the memory allocated is the dimensions multiplied together along with information tables used to form the array. Say, an array is x(10,10), the memory used will be 10X10 = 100 along with the header information. The "size" command can tell you how much memory is used and how much is available. The *expunge* command will free up variable space taken up but arrays.

end Define logical end of program. Causes termination of current run. The same as "stop", except that stop displays the line number where it stopped, whereas "end" just stops the program. *End* is more desirable when you are writing output data to a file and you don't want interpreter messages to appear in the output file.

expunge Force all variable space, including subscripted variables to be freed. Or deallocate used variable space.

files Displays the currently open file or loaded BASIC program file, (if any) and any files open for read, read, write or append. The highest number of files allowed to be open in any mode or combination of modes is 8.

for- next Cause code enclosed by this combination to be executed under the conditions specified in the *for* statement as in: *for <variable> = <expression1> to <expression2>.* The step directive, tells the *for* loop how much to step for each iteration as in: **for x = 1 to 360 step 2**.

gosub line# *Goto* a subroutine, resume from following statement after *return* encountered.

goto line# Force execution to continue starting at the line# specified.

If (expr1) relational (expr2) then line#

Redirect program flow to line# if expr 1 is related to expr2 by the specified relational. The then in the then statement can be optionally replaced with goto or gosub. The if statement can also take the form:

if (expr1) relational (expr2) then var = (expr) (form 1)

if (expr1) relational (expr2) [(boolian) (expr3) relational (expr4)] then [line number]
if-then-else-endif When no line number is given, the “structured” if is assumed.

```
100 if a > a-b && c > c/d then
200     print 200
300 else
400     print 400
500 endif
```

If-then-else constructs can be nested.

```
1100 print "Type 1 for if-then-else test or 2 for logic test";
1200 input n
1300 on n goto 1400,3900
1400 print "if-then-else test"
1500 print "Input numbers for a,b,c,d";
1600 input a,b,c,d
1700 if a > b then
1800    if c > d then
1900            print "a > b and c > d"
2000    else
2100            print "a > b and c < d"
2200    endif
2300 else
2400    if a = b then
2500            print "a = b"
2600    else
2700            if c > d then
2800                    print "a < b and c > d"
2900            else
3000                    if c = d then
```

```
3100                              print "a < b and c = d"
3200                     else
3300                              print "a < b and c < d"
3400                     endif
3500            endif
3600    endif
3700 endif
3800 goto 1400
3900 print "Logic test,  Input combinations of 1 and 0 like"
4000 print "0,0 -- 0,1 -- 1,0 -- 1,1";
4100 input a,b
4200 if a && b then 4600
4300 if a || b then 4700
4400 print "false"
4500 goto 4100
4600 print "a and b is true"
4700 print "a or b is true"
4800 goto 4100
```

Double relationals form:

```
1100 print "input a,b,c,d";
1200 input a,b,c,d
1300 if a < b && c < d then
1400 print 1400
1500 else
1600 print 1600
1700 endif
1900 if a > b && c < d then
1910 print 1910
2000 else
2100 print 2100
2200 endif
```

input [#flldes 1var1[,var2,var3.....] Input assigns the number values typed at the input prompt, to the variables specified in the command line, which can be a single letter variable or an array variable, such as: line number input a, b, c(2,2,2). Note that if fewer numbers are entered at the input prompt '?' than the operand or command line variables, the input command will display a message telling the user that more values are specified or type the letter 'q' to quit. It will continue to ask until the number of comma separated variables like, 20, 3.14, 98.6 is equal to or more than the number of variable given in the input argument. If extra entries are made, they will simply be ignored. Input is also the method of reading variables from a file into the BASIC program. In the file, the input numbers are arranged in rows of comma-separated numbers. If the number input numbers in the row is less than the number of variables expected, an error will be issued, suggesting that the user fix the input file and the BASIC program will rewind the file and stop.

let variable = expr Assign the value of expr to variable.

list [lownum [,highnum]]

List the text in working storage. If lownum is given then only that number is listed., if lownum and highnum are specified, then a listing is displayed between the given statement numbers.

load [program name].

Same as the *old* command, except working storage is not cleared.

mov startnum, endnum, newnum [,increm]

The *mov* command causes the lines, beginning with startnum and ending with endnum to be moved (ie. resequenced) to the line beginning with newnum and incremented by increm. The default value for incrern is 10. All references to the moved lines are updated. The' user is responsible to see that line numbers associated with moved lines do not conflict with existing lines which will cause loss of program text. *mov* is similar to *reseq* (see below) except that only the specified lines are resequenced.

n The n command lists 20 lines at a time. To continue listing, type <ENTER>. To exit, type 'q'.

new Clear program working storage for new program to be typed.

old [program name] or open [program name]

Clear user space and load program. If old is typed with no argument it will prompt the user for a program name if not defined or load the last defined program name.

on (expr) goto line#, line# ......

Is a selective *goto* with multiple line number targets. The target branched-to depends on the value of expr which is truncated. Control is passed to the first line# specified after *goto* if the value of the expression is 1. Control passes to the second line# if the value is 2, the third if 3 and so on.

on (expr) gosub line#, line#.......

Same action as on-goto, except action is that of gosub.

pause Causes execution to be suspended until a "newline", or "return" is typed. This is useful for programs which need to be continuously in run, but need to allow a time for user action i.e. unit insertion. **Typing the letter 'q'** while paused, will stop execution or quit running the BASIC program.

print [#fildes] (expr's, quoted strings or tab operators)

The print statement is a limited format display statement in which expressions are evaluated and displayed along with quoted literals. The **tab**(expr) operator causes the print head to move to the absolute column position computed by expr provided the current head position is smaller. The specifiers must be separated by one or more commas or semicolons. **hex**(expr) prints the hex converted value of expr, but, unlike **oct**(expr), hex() is not a math function, but only a print function, because hex numbers in this interpreter cannot be handled by the (double) values of this interpreter. Note that the results of base conversions are REPRESENTATIONS of the converted print in floating point.

prec[ision] <expr> = 1 through 11. Sets the precision of floating numbers in the print command. Where 1 – 10 represent the number of decimal places and 11 is automatic

printf (format string) [,exprl,expr2 .... expr10]

This is an interpretive implementation of the 'C' programming language function, printf. It is, however restricted to only the floating point format control specifiers 'f' and 'g'. Use of any of the other specifiers such as 'o', 'd' or 's' will give erroneous results. Print controls such as \b (backspace), \n newline), \r (return) or \t can also be used. The printf format was chosen in lieu of the. usual print using command because it was felt that printf is not only a 'C' language standard but easier to use than print using. Usage Example:

```
100 printf[ #n]War a=%2.2f\t Var b=%g.\n" ,a, b
```

Randomize(seed) Causes md statement to start at an "unpredictable" value. Etime() can be used to seed the random number as time (in seconds since midnight Jan 1, 1970 is continually changing. Etime is covered later.

read varl,var2,var3 ....

The read statement causes data to be assigned to each variable in the list from the constant

rem The remark statement causes no operation in BASIC but may be followed by any string of characters for the purpose of commenting a program. They are of paramount importance in making a program readable by human beings and are, therefore, strongly recommended. As in:

rem //////// This is a comment /////////

// does the same thing as *rem*, but added to make comments less cumbersome looking as in:
// ///////////// This is a comment ///////////
Note that a space must follow the double slash, otherwise the interpreter will see the whole field as a BASIC command.

Renum[ber] [startnum [, incremi]

The renumber command causes the statement numbers and all references to them (such as ifs gotos, gosubs, etc.) to be renumbered starting at startnum and incremented by increm. If startnum and/or increm are omitted, the default values are 10 and 10 respectively.

Restore Restores the data pointer to the first field of the first data.

return Return from subroutine called by gosub statement.

run [program name]

Run basic program specified. If no argument is given, run attempts to execute whatever is currently in working storage.

s line#/old-string/new-string/

Substitute in line line# the new-string for the old-string. The last delimiter is optional, unless new-string is null in which case it is desired that oldstring be removed.

old-string merely be removed. See the undo command.

sing [line#] Enter the single step mode starting at the line# specified or at the first

line of the program if no line# is specified. In single step mode an instruction is executed and then the prompt " is displayed. At this time the user may enter any command (i.e. print) or hit the "return" key to execute the next instruction. See the con command.

size Causes amount of storage used and remaining or free space in decimal number of bytes.

stop Stop execution of program, giving the line# of the stop command. The end command does the same thing but without the message.

save [program name]

Save the contents of working storage specified by program name. If no program name is given, it referenced file-name is used. If no file name was referenced, the user is prompted for a name. The default load and save folder is the one that this executable resides in. Otherwise, a complete path or one relative to the folder that this executable resides in as in:

save filename, *old* or *load* filename or save c:\<folder>\filename

undo Undo last s command or single line deletion

STRUCTURED FLOW COMMANDS

if-then-else-endif Allows more control The file the test in if directs to two different possibilities, condition met or condition not met. Gotos or on gotos are not allows within a structured if-then-else scope.

break can be used to break out a loop like for-next before the final condition is met

continue forces continuation of for loop before loop condition is met

See example for-next-break-continue example.

3.2 File Commands

The file commands: append, *openin*, and *openout* are followed by one or more file-names separated by commas. Files are assigned to designators (integer values between 1 and 8 inclusive) in the order that they are open. All commands such as print and input which refer to a file use the designator number preceded by a character to refer to that file as in:

```
100 print #1"hello world" or 100 input #3a (x,y)
```

append file1,file2 .... file4l

If file exists open for output cause new data to be appended. If file does not exist, the named file is created.

openin file1[,flle2 .... file4l

Open file for input.,. File must exist. -

openout fllel1[,flle2 .... file4]

Create, named file(s) and open for output. If named files exist, the old. data is destroyed.

closef #fildes . .

Close file associated with file designator.

closeall Close all files input and output.

files Displays the currently open file or loaded BASIC program file, (if any) and any files open for read, read, write or append. The highest number of files allowed to be open in any mode or combination of modes is 8.

3. Functions

3.1 Standard Functions

abs(expr)	Absolute value.
atn(expr)	Arc-tangent.
cos(expr)	Cosine.
exp(expr)	Natural exponential.
Int(expr)	Integerize, or truncate fractional part of rs4�f exp.r.
log(expr)	Natural log.
rnd(expr)	Return random number between 0 and evaluated expr.
Sin(expr)	Sine.
sqr(expr)	Square root.
etime(1)	epoch_time number of seconds since midnight January 1, 1970.

4. Modes of Operation

4.1 Editor or Idle Mode

When the BITE interpreter is invoked with no argument, a prompt " appears meaning that the interpreter is waiting for the user to enter something from the keyboard. BITE is then said to be in the Editor or Idle mode.

Editing is accomplished as it is in any, BASIC language interpreter in that lines are entered by typing a line-number followed by the statement and removed or deleted by merely typing the line-number. Listing is accomplished with the list command (explained under "Standard Commands"). In addition to the above, it is possible to list single lines by typing the retun'key in which case the program is listed one line-at-a-time, starting at the first. When the last one is reached, the sequence starts at the first line again. At any time it is also possible to type the symbol to "backup" a line-at-a-time. Other editing facilities are s delete, and reseq also explained under "Standard Commands".

4.2 Run Mode

If the run command is typed and a program is currently in user storage, the program begins execution, starting with the first line of the program, then executing each line in order of line numbered sequence. The sequence of execution is altered by program flow control statements like jr, for-next or any statement, containing a goto.

4.3 Immediate Execution Mode

Immediate execution is accomplished by typing a command without preceding it with a line number. Although this is possible with all commands, it doesn't always make sense. For example, using commands that control program flow in immediate mode is unlikely and often disastrous.

Immediate mode is designed so that the user may get immediate action as in the command *run* or *print* a. Some commands are almost always used in immediate mode such as *q, delete, expunge, load, list, old, renum, save, etc.*

4.4 Single Step Single step mode is entered with the sing command and exited with, the con command. During this mode,' one may find "BUGS" in the program by observing the program flow or sequence or examining the values of variables at given points in the program to see if they have the expected values. See *sing* or *con* under the "Standard Commands" section of this paper.

5. Interruption of program Doesn’t work, yet

At times it becomes necessary to escape from an endless loop or abort an action such as list before it completes. To cause such an interruption, the (DEL) or (RUB) key is typed.

6 Error Messages

Diagnostic error messages are issued by the interpreter which indicate syntax errors , system failure, illegal commands or expressions, etc.

6.1 Standard Error Messages

NUMBER	**MESSAGE TEXT**
0	REFERS TO A NON-EXISTING LINE NUMBER
1	UNRECOGNIZABLE OPERATION
2	CANNOT OPEN FILE
3	ILLEGAL VARIABLE NAME
4	BAD FILENAME
5	WORKING STORAGE AREA EMPTY
6	RUNS NESTED TOO DEEPLY
7	UNASSIGNED VARIABLE
8	EXPRESSION SYNTAX
9	BAD' KEYWORD IN STATEMENT
10	IMPROPER OR NO. RELATIONAL OPERATOR
11	UNBALANCED QUOTES
12	FILE EDITING NOT PERMITT ED IN SINGLE STEP MODE
13	MISSING OR ILLEGAL DELIMITER
14	GOSUB- WITH NO RETURN
15	IS. FATAL
16	UNBALANCED PARENTHESIS
17	UNKNOWN MATH FUNCTION.
18	NEXT WITH NO OR WRONG FOR IN PROGRESS

19	CANNOT PROCESS, IMAGINARY NUMBER
20	WHAT?
21	BAD' DIMENSION SYNTAX
22	TOO MANY DIMENSIONS
23	REDUNDANT DIM STATEMENT
24	NOT ENOUGH WORKING STORAGE SPACE
25	VARIABLE NOT DIMENSIONED
26	WRONG NUM ' OF DIMS
27	ONE OR MORE DIMS LARGER THAN ASSIGNED
28.	NEG. OR ZERO DIMENSION. ILLEGAL
29	DIVIDE BY ZERO
30	BAD TAR SPEC. INPRINT
32	BAD FILE DECLARE SYNTAX
33	OUT OF DATA
34	FILE-NAME TOO, LONG
35	FILE DES. USED UP
36	FILE NOT OPEN FOR OUTPUT
37	FILE NOT OPEN FOR INPUT
38	EXPRESSION YIELDS AN IMPOSSIBLE VALUE
39	PRINTF: ARG COUNT MISMATCH
40	PRINTF: MORE THAN 10 ARGS
41	LINE TOO LONG FOR STRIP PRINTER
42	MOV REQUIRES 3 LINE #'s, SPACING IS OPTIONAL
43	BAD NAME OR LINE NUMBER AT BEGINNING OF SUBROUTINE
49	POSSIBLE EMPTY LINE IN INPUT FILE

EXAMPLE BASIC PROGRAMS

These can be cut and pasted into a Windows notepad file, saved as a .txt then loaded into BASIC. Making sure that there are no intervening or trailing blank lines.

Fahrenheit to Celsius table 0 to 101 c

```
2000 openout temptable.txt
2100 prec 2
2200 dim f1(51)
2300 dim c1(51)
2400 dim f2(51)
2500 dim c2(51)
2600 for n = 1 to 51
2700 let c = n-1
2800 let f1(n) = 1.8*c+32
2900 let c1(n) = c
3000 next
3100 for n = 1 to 51
3200 let c = n+50
3300 let f2(n) = 1.8*c+32
3400 let c2(n) = c
3500 next
3600 printf "\n  F        C                 F        C\n"
3700 printf "-------------------------------------------------\n"
3800 printf #1"\n        F        C                 F        C\n"
3900 printf #1"-------------------------------------------------\n"
4000 for n = 1 to 51
4100 printf "     %3.2f    %3.2f    |        %3.2f    %3.2f\n",f1(n),c1(n),f2(n),c2(n)
4200 printf #1"  %3.2f    %3.2f    |        %3.2f    %3.2f\n",f1(n),c1(n),f2(n),c2(n)
4300 next
4400 closef #1
```

```
1000 rem //         3 X 3 Determinant solution
2000 rem //    Solution to 3 variable simultaneous equations
2100 rem
2200 rem //                 a11X + a12Y + a13Z = b1
2300 rem  //                a21X + a22Y + a23Z = b2
2400 rem //                 a31X + a32Y + a33Z = b3
2500 rem
2600 print "Input values for b1, b2, b3 - like: 10,20,-30"
2700 input b1,b2,b3
2800 dim a(3,3)
2900 read a(1,1),a(1,2),a(1,3),a(2,1),a(2,2),a(2,3),a(3,1),a(3,2),a(3,3)
3000 gosub 6200
3100 let d = a1-a2+a3
3200 print "d = ";d
3300 restore
3400 read a(1,1),a(1,2),a(1,3),a(2,1),a(2,2),a(2,3),a(3,1),a(3,2),a(3,3)
3500 read a(1,1),a(2,1),a(3,1)
3600 gosub 6200
3700 let d1 = a1-a2+a3
3800 print "d1 = ";d1
3900 restore
4000 read a(1,1),a(1,2),a(1,3),a(2,1),a(2,2),a(2,3),a(3,1),a(3,2),a(3,3)
4100 read a(1,2),a(2,2),a(3,2)
4200 gosub 6200
4300 let d2 = a1-a2+a3
4400 print "d2 = ";d2
4500 restore
4600 read a(1,1),a(1,2),a(1,3),a(2,1),a(2,2),a(2,3),a(3,1),a(3,2),a(3,3)
4700 read a(1,3),a(2,3),a(3,3)
4800 gosub 6200
4900 let d3 = a1-a2+a3
5000 print "d3 = ";d3
5100 restore
5200 data 6,5,9
5300 data 2,0,1
5400 data 3,4,0
5500 data b1,b2,b3
5600 let x = d1/d
5700 let y = d2/d
5800 let z = d3/d
5900 print "x = ";x;" y = ";y;" z = ";z
6000 end
```

MAXIMUM FILE OPEN TEST WITH DEMONSTRATION OF FILES COMMAND
CLOSES TWO FILES THEN OPENS TWO FOR INPUT

```
90 rem //    Tests file open for read or write up to the maximum of 8 files
95 rem //    Opens 8 files, closes two, then opens two for input
96 rem //    Demonstrates the files command to show how the file table
97 rem //    is filled each time files are open or closed
98 rem //
100 openout aout.txt,bout.txt,cout.txt,dout.txt,eout.txt,fout.txt,gout.txt,hout.txt
200 print #1"Test 1"
300 print #2"Test 2"
400 print #3"Test 3"
500 print #4"Test 4"
600 print #5"Test 5"
700 print #6"Test 6"
800 print #7"Test 7"
900 print #8"test 8"
950 print "All file slots open for output."
1000 files
1100 closef #2
1200 closef #7
1250 print "Slot 2 and 7 closed."
1300 files
1400 openin test1.txt,test2.txt
1500 print "All input and output files closed.  Only current BASIC file, currently open, if any."
1550 print "Two files open for input to fill empty slots 2 and 7."
1600 files
1700 closeall
1800 files
```

TEST AND DEMO OF PRINTF COMMAND

```
2000 let p = 3.1415926535
2200 let e = 2.71828
2300 let t = 98.6
2400 let a = 222.95
2500 printf "p = $%3.2f e = $%3.2f\tt = $%3.2f\n", p,e,t
2600 printf "p = $%3.3f e = $%3.3f\tt = $%3.3f\n", p,e,t
2700 printf "p = $%3.2f e = $%3.2f\tt = $%3.4f\n", p,e,t
2800 printf "p = $%3.2f e = $%3.2f\tt = $%3.5f\n", p,e,t
2900 printf "p = $%3.2f a = $%3.2f\tt = $%3.5f\n", p,a,t
```

TEST AND DEMO OF NESTED IF-THEN-ELSE WITH TAB INDENTS

```
1100 print "Input numbers for a,b,c,d";
1200 input a,b,c,d
1300 if a > b then
1400     if c > d then
1500             print "a > b and c > d"
1600     else
1700             print "a > b and c < d"
1800     endif
1900 else
2000     if c > d then
2100             print "a < b and c > d"
2200     else
2300             print "a < b and c < d"
2400     endif
2500 endif
2600 goto 1100
```

FOR-NEXT-BREAK-CONTINUE

```
1000 for x = 1 to 10
1500     if x > 5 then
1600             break
1650     else
1700             print x
1710     endif
1800 next
1900 print "breaked at ";x
2000 for x = 1 to 10
2100     if x > 5 then
2200             continue
2300     else
2400             print x
2500     endif
2600 next
2700 print "continued to ";x
```

BASIC PROGRAM TO GENERATE SIN, SQUARE, SAWTOOTH WAVES GIVEN THE NUMBER OF HARMONICS THE FOURIER SERIES CALCULATRS OUT TO

```
90 expunge
100 print "Input number of Harmonics";
200 input n
300 if n = -1 then 1700
400 print "Select waveform: 1 for squarewave, 2 for sawtooth, 3 for triangle";
500 input w
600 if w <= 3 then 1200
700 print "Number ";w;" is out of range, input 1 - 3"
800 goto 400
900 rem //////////////////////////////////
1000 rem ///Print a square wave in asterisks
1100 rem //////////////////////////////////
1200 let a2 = 720
1300 for a = 1 to a2 step 40
1400 on w gosub 1800,3700,4700
1500 next
1600 goto 100
1700 stop
1800 rem square_f
1900 let y = 0
2000 let u = 0
2100 let s = 2
2200 for h = 1 to n step 2
2300 let u = u+((1/h)*sin(h*a))
2400 next
2500 rem Multiply Series with amplitude of 10
2600 rem Add a :carrier of 15 so that negative values
2700 rem don't get clipped
2800 let y = 10*u+15
2900 gosub 3100
3000 return
3100 rem plot_star
3200 for x = 1 to y
3300 print " ";
3400 next
3500 print "*"
3600 return
3700 rem sawtooth_f
3800 let y = 0
3900 let u = 0
4000 let s = 2
```

```
4100 for h = 1 to n step 1
4200 let u = u+(1/h)*sin(h*a+(h%2*180))
4300 next
4400 let y = 10*u+15
4500 gosub 3100
4600 return
4700 rem triangle_f
4800 let y = 0
4900 let u = 0
5000 let s = 2
5100 for h = 1 to n step 2
5200 let u = u+(n/fact(h))*cos(h*a)
5300 next
5400 let y = 10*u/h+20
5500 gosub 3100
5600 return
```