SCIENTIFIC PROGRAMMING

with

TRUE BASIC

SUBJECT OUTLINE

by

D. W. McClure
Professor Emeritus
Chemistry Department
Portland State University

SCIENTIFIC PROGRAMMING

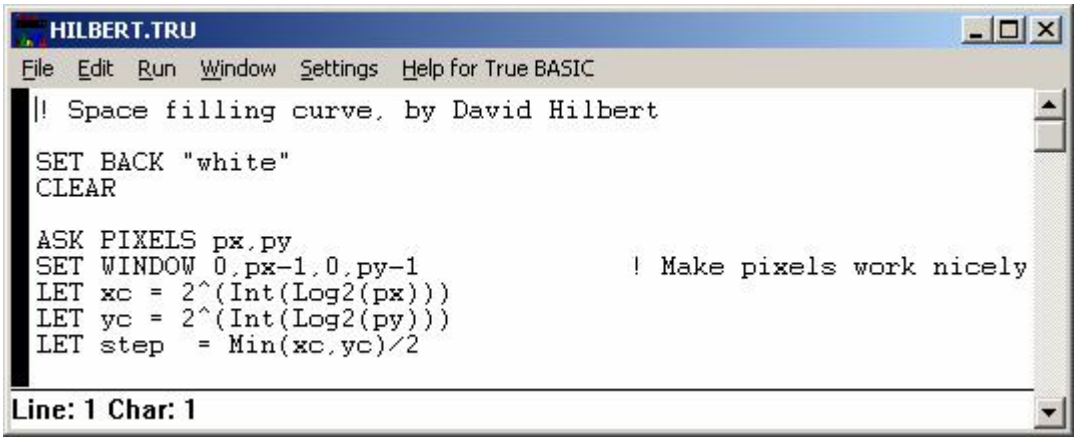
Professor D.W. McClure, Chemistry Department
Portland State University

ASSUMED BACKGROUND FOR CH 443

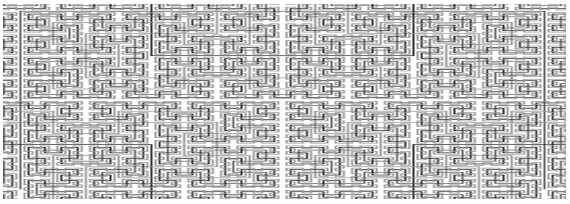
It is assumed that you know your way around a computer and in particular, MS Windows 2K or XP. You should know how to use MS Word, MS Excel, Windows Explorer and Internet Explorer as well as know how to copy programs to and from the computer, how to delete files, save files and so on. In short, we are not going to teach computer basics. On the other hand, it is NOT assumed that you have ever done any programming so this course starts at the beginning. It is also assumed that you have a copy of True Basic (any version) available on your computer.

GETTING STARTED

Start the True Basic program by double clicking on the True Basic icon. You should now see a typical '**Open File**' dialog box as well as a Command Window. Double click on TB Demos, scroll through the list of programs available and select the File 'Hilbert'. Either double click the file name or click **Open**. Note that the main window containing code now opens. This main window is called the **Source** or **Code Window**. In addition, note the menu bar across the top. This menu bar contains all of the commands you will need to edit, run and save programs. The screen should look like:



Now click on **Run** and watch what happens. You should see a new window (called the **Output Window**) containing something that looks like (only much larger and in color):



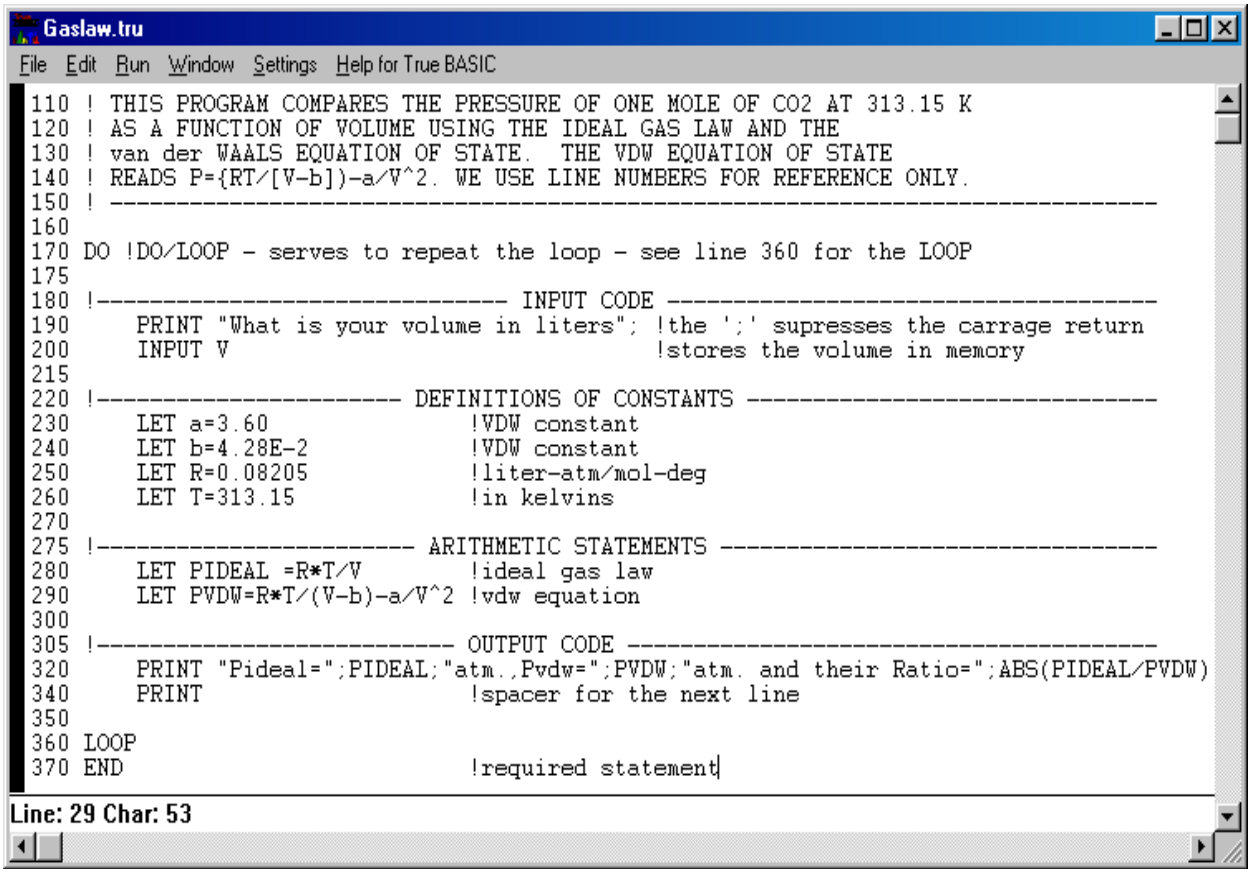
Congratulations, you have just run your first TB program. You can dismiss the Output Window by first clicking on **Menu, Stop** and then clicking anywhere on the Window. When you close the program, you will get the non-standard Window's dialog box that looks like:

Now click on **Discard** since you don't want to **Save** the Hilbert program. In general, you want to be very careful at this stage when writing programs because *inadvertently clicking **Discard** when you meant to **Save** can cost you a lot of work and time because there is no going back if you made a mistake!*

Now that you have loaded and run a True Basic (TB) program, let's take a look at how code works.

AN EXAMPLE PROGRAM

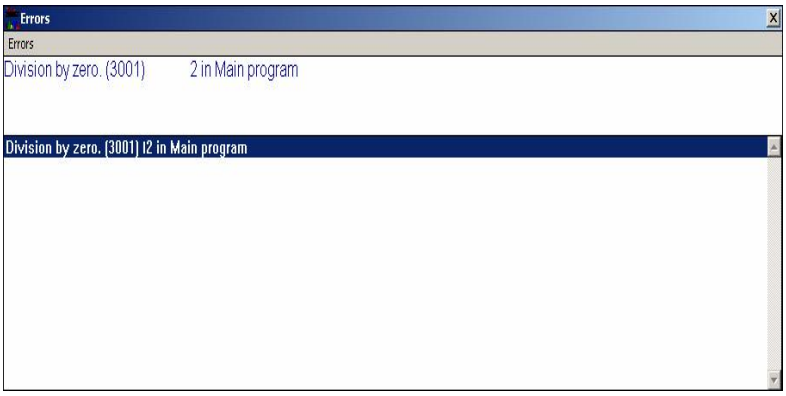
Probably the easiest way to get a feel for a program is to see and analyze one. The following short program calculates the pressure of one mole of CO_2 gas at 313.15 K as a function of volume using the ideal gas and the van der Waals equations of state. The van der Waals' equation reads $P= RT/(V-b)-a/V^2$. A screen shot of the program listing is given below (Code Window) as well as typical output (Output Window) when run.



This code listing illustrates a couple of important points that all programs have in common,

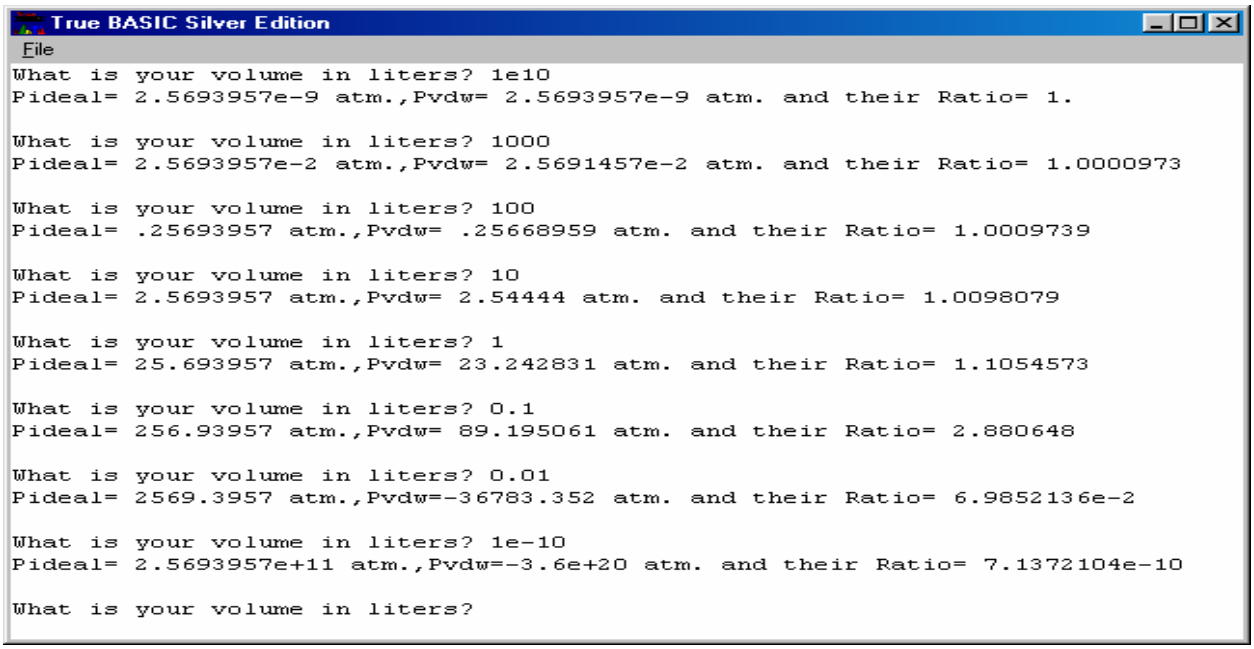
1) Serial Execution: i.e., one line at a time starting at the top and progressing line by line through the code until a branch point is encountered where some kind of decision is made. Depending on the decision, the program flow may jump to another portion of the program, execute new code and then return to the branch point. If there is no branch point (and there is not one in the Gas Law Program), each line will be executed until eventually the END statement is reached.

2) Key Words: all program languages use a set of 'key' words that characterize that language and are the statements that make things happen. For example, the words **REM** (remark), **LET**, **PRINT**, **INPUT**, **ABS** (absolute value) and **END** are but a few of the key words that characterize the True Basic language. Each one has a specific meaning and must be used, not only in the correct problem solving context, but also must be absolutely correct in spelling and construction. Computer programs are insanely single minded in insisting that the code structure be absolutely correct or the program will not run in which case you will get what is called a ***'syntax error'***. *Expect to become very familiar with syntax errors*. Syntax errors are delineated in an **Error Window** in True Basic as in the following example of a 'division by zero' error.



These errors are the best kind to have because their existence is obvious since the compiler will complain, the program will not run and usually the problem is simple to fix. Far more pernicious are errors of a logical nature because these errors lead to ***wrong results*** even though the program runs fine. In other words, ***the fact the program runs does NOT mean the output is correct!*** More about this in a moment.

Begin by entering the GasLaw code, exactly as written, into the code window (remembering to press return at the end of each line). Once you have finished typing in the code and ***before*** you try and run the program, save it to ***your*** disk (call it GasLaw or something else that will help you remember the nature of the program). Why? Because it is more than a little possible that you have made such an egregious error in inputting the code that you will crash True Basic and then guess what – you have to start over. Next obtain a listing of the code by clicking on **Print** in the File Menu. Now you are ready to run the program.



To do so, click on **RUN** in the tool bar. The program will respond by opening the Output Window (don't run it full screen as it will hide the Error Window) and displaying the prompt 'What is your volume in liters' with the cursor patiently waiting after the question mark for a reply. You then type, for example, 50 and press return. The computer then prints the phrases in quotes after the PRINT statement (code line 320) followed by the computed results. Now click on the File Menu in the tool bar of the **Output Window**. The menu has three choices, **Print**, **Copy** and **Stop**. **Print** sends the output to the printer for a hardcopy, **Copy** copies the output to the clipboard and **Stop** halts program execution. You will need to click on **Stop** in order to end the program; otherwise it will continue to prompt you for new 'volumes'. To get a hardcopy of the program code itself, use the **PRINT** command in the **File** Menu of the **Code Window**.

Now try running the program and enter 0 after the volume prompt. The program crashes (does not run) and gives you a 'Division by zero' error. An error of this type is always 'fatal' – not to you, but to the program. You might think a little about the logic you would use to stop such an error from ever occurring in terms of program code. Note: if you don't see the Error Window, it's under the Code Window.

Wrong input data will behave the same way - the program runs but the results are meaningless. Hence the phrase, garbage in = garbage out. These non-fatal errors are pernicious because they rarely announce themselves in an obvious way thus requiring you to ferret them out. The rule that must be paramount for any programmer is that:

ALL NUMERIC OUTPUT MUST BE VERIFIED

How? - By the simple expedient of doing an independent hand calculation of at least one value. If loops are involved where a number of values are computed then there should be an independent verification using at least three values -- the least computed value, the largest computed value, and some value in the middle. Only then can you feel reasonably assured that your output is valid.

Finally, after all logical and input errors have been eliminated and the program runs and gives 'correct' results there still remains a potential for additional computational errors due to propagated computer round off and/or truncation errors. These errors, which are more subtle in origin are outside the scope of this manual.

Verification of the Gas Law Program is simple enough. Pick a couple of volumes that represent the range of possible values of interest, and compute, by hand, the corresponding pressures and their ratios. Do not, however, use the program code for the definitions of the equations, rather, re-solve for the variables of interest after verifying the correctness of the equation from an independent source. Why? - because you will never know if you have coded the 'equation of state' in the program incorrectly if you use the program definitions.

3) You will notice that the Gas Law Program consists of basically three parts, a part where you input the data (the volume in this case), a part where you output the results (print the pressures and ratio) and the 'stuff in between' where the 'stuff in between' usually determines the complexity of the code since input and output is usually pretty straight forward.

We now begin a detailed study of the True Basic language. Before doing so however, let's take a quick look at the PC screen because its characteristics determine how our output looks.

Long Code Lines

It is quite likely that you will occasionally write a line of code (usually an arithmetic statement) that will exceed the length of the screen. This does not present a problem since moving the cursor to the right on that line will scroll the line so that you can see it. However, when the code is listed, you will not see that part of the line that exceeds the screen width. A useful solution is to split a long line into two or more lines so scrolling is unnecessary. This is done by typing a "&" at the point where the line is split and at the beginning of the next line. This will obviate the need to scroll and a program listing will show exactly what you see on the screen

THE TRUE BASIC LANGUAGE

Numeric and String Constants

True Basic implements two kinds of constants - numeric constants and string constants. A **numeric constant** is just a number and consists of combinations of the digits 0,1,...,9, a decimal point and a + (implied) or - sign. Thus 5 and 6.022e23 are numeric constants.

The 'E' or 'e' notation in the last case stands for 6.022×10^{23}. The number -1e-123 is valid but e-123 is not (a number must precede e) and neither is -1×10^{-123}, i.e., you *must* use the e notation for powers of 10.

A ***string constant*** is any combinations of characters, i.e., letters and/or numbers enclosed between quotation marks, such as: "Computer", "6.022e23" or this paragraph, if enclosed by quotation marks. The quotation marks serve only to identify the string as a string and are not part of the string itself. Note that "6.022e23" is a string, that is, a sequence of characters, and must not be interpreted as a number. As we shall see, numeric constants and string constants are stored and manipulated quite differently by the language.

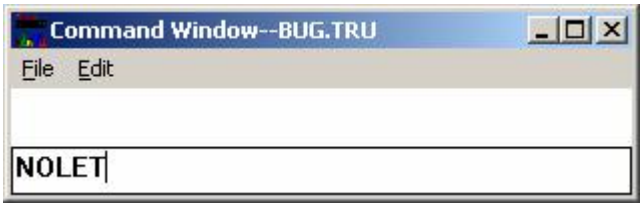
Numeric and String Variables
Just as there are two kinds of constants in True Basic, there are also two kinds of variables, numeric and string. ***A numeric constant can only be assigned to a numeric variable and a string constant can only be assigned to a string variable***.

Let's now look at the mechanics of how assignments are made, beginning with numeric variables.

THE ASSIGNMENT STATEMENT

A) Numeric Variables and Numeric Constants
Returning to the Gas Law program and the expression R = 0.08205, we see that every assignment consists of the following four parts.

1) The Assignment Operator -- **LET**
The optional key word **LET** is a True Basic anachronism used to assign values to variables, both numeric and string. It is not required, but to ignore it requires the **NOLET** option which you type into the textbox in the **Command Window** – and ***remember to press Return***. True Basic is the only language still using this key word. In this manual, we will NOT, for the most part use **LET** as it is unnecessary, annoying and specific to True Basic only. Here is what the **Command Window** looks like with '**nolet**' in the text box.



2) The Variable Name
In this case, and in keeping with the usual notation, we have used R to represent the constant 0.08205. However, variable names can be up to 31 characters in length, can contain numbers (but not as the first character) as well as letters and the underscore '_'. No other symbols are acceptable including spaces.

Case Sensitivity
True Basic is case insensitive, that is, code written in lower case, upper case or a mix of the two is identical from the compilers point of view. Thus the variable name SumOfSquares, sumofsquares or SUMOFSQUARES is the same in all three cases. Languages like C or C++ would treat these as three different variables.

The construction of a variable name is a personal choice but they really should reflect what the variable does or represents. For example, if we had a variable to represent the time of day in some piece of code, we might write Time_of_Day, or TimeOfDay for the variable name. Likewise, if we were doing a statistical calculation where we needed the sum of the squares of the residuals we might use SSResiduals, SS_of_Residuals etc. What we would not do is use a name like x for this variable because, by itself, x conveys nothing about the variable. On the other hand, a name like Sum_Of_Square_Of_The_ Residuals is just too masochistic if one has to type the name of the variable very often.

3) The Right Hand Side of the Assignment Statement
The right hand side of the expression can be a number (e.g. 0.08205, i.e., a numeric constant) or an expression containing other variables (more about that in a moment). Numeric constants, as we have seen consist of the usual 10 (0,1,2...9) digits, a decimal point if needed, possibly a sign as in + or - and, when needed, notation to denote powers of 10. Commas, spaces, dollar signs etc. are not permitted.

Thus the choices:

X = -2.3,
NumberOfTextLines = 0
TestDigits = +1.23456789
Avogadros_Number = 6.022E+23

are all valid assignments. Note that you cannot have spaces in the variable name – hence the use of the underscore in Avogadros_Number. Note again that in order that we can ignore the LET statement, we have to use the NOLET option in the Command Window.

4) The Equality Sign and the Meaning of the Assignment Statement
The '=' sign is not to be interpreted as an equals sign but rather as an assignment operator.

To help clarify this concept, consider the line of code from the Gas Law Program; R=0.08205

Here we are assigning the *numeric constant* 0.08205, the gas constant, to the *numeric variable* R. If one did not know better this might look like a simple algebraic statement, but from the view point of the computer, this statement sets up a location or address in computer memory that symbolically represents both the variable name, R and its value, 0.08205. Every one of the six assignments made in the gas law program corresponds to a different memory location where each location represents both the name of the numeric variable and its current value.

Another analogy is to think of the memory location for R as a mail box with the name R and the contents of the mail box as the value R has at the moment, i.e., 0.08205. In this sense then, the '=' sign does not denote 'equality', but rather should be viewed as an 'assignment' operator which says 'assign the value on the right hand side of the expression to the memory location representing R.

Because '=' is not to be interpreted as an equality sign, statements like

```
Count = Count+1
New Value = Current Value + 1
```

make sense. This statement says 'go to the mailbox (location) in memory with the address representing the variable 'Count', open the box, select the current content which is a number, add 1 to it and then store the new value in the Count mailbox so that Count(new value) = Count(old value)+1.

For example, suppose Count is initially 5. The assignment operation says, go to the memory location representing Count, add 1 to the current value, which is 5, delete the old value of 5 from memory and finally replace it by the new value 6. Because it is the same memory location representing Count, the value of Count now has the value 6, i.e., same mailbox, new contents. This example should also illustrate that numeric variables really are variables in the sense that they can take on new values any time the program requires them to do so.

Statements like Count = Count + 1 are called counters because they literally count how often a section of code executes and are used routinely in programming. This statement is also an example of an arithmetic statement or expression.

Arithmetic Statements or Expressions
If all one could do in a program is assign numbers or constants to a variable name, computer programs would be of little value. However, assignments like:

```
Area = PI*R^2                          ! PI is 3.14...
V = n*R*T/P
Root1 = (-b + SQR(b^2-4*a*c))/2*a   ! SQR is the square root
WordCount = WordCount + N
```

are acceptable code statements, and make sense if we remember that each variable in these expressions is just a memory location to which is assigned a numerical constant. The Gas Law Program used two assignments of this type, namely,

```
IdealGasPress = R*T/V
VdwGasPress = R*T/(V-b)-a/V^2
```

Obviously, an arithmetic expression cannot be evaluated unless there actually is a number representing each variable stored in memory. For that reason, it was essential that these two expressions go ***after*** the assignment of a, b, R, T and V in the program. In fact, a, b, etc. **do** have values before they are assigned

values in the program, namely, 0, and this is because True Basic automatically initializes all variables to 0 when the program is run. You can verify this by running the following program.

```
y = 1/x
x = 5
PRINT "y = ";y
END
```

You will get a 'division by zero' error because x has been initialized to 0 at runtime. The remedy is simple, just move x = 5 to the line above y = 1/x and re-run the program. This may seem all pretty obvious, but errors due to the placement of arithmetic statements relative to their variable assignments are a common problem, especially when one is just learning to program. Remember, compliers do not 'look ahead' to see what's missing.

Let's now look at the machinery for constructing arithmetic expressions.

Operators for Constructing Arithmetic Statements
The following arithmetic operators are available to link variables when building equations.

ARITHMETIC OPERATORS

Operator	Meaning
+	addition
-	subtraction
*	multiplication
/	division
^	exponentiation

In addition to the five arithmetic operators, there is the parenthesis ( ) which is used to separate operators, define blocks of expressions and determine the order in which operations are carried out. Parentheses can be nested as deeply as needed.

In an expression containing multiple operators, the order in which these operators are executed is important. Ignoring this fact is the cause of a lot of grief when the program runs but gives incorrect results. The following table illustrates the order in which the arithmetic operators are evaluated.

OPERATOR PRECEDENCE

Order	Operation
1st	expressions in a parentheses
2nd	exponentiation
3rd	multiplication/division
4th	addition/subtraction

Operations like multiplication and division or addition and subtraction have equal priority and are executed left to right in the order they are read by the compiler. Thus x/y* z is evaluated as (x/y) * z, not as x/(y* z).

The following table shows how the order of execution can affect results

ORDER OF EVALUATION IN NUMERIC STATEMENTS

Arithmetic Expression	Compiler Evaluation	Result
3+5*6	3+(5*6)	33
(3+5)*6	(3+5) *6	48
2+3/4+2	2+(3/4)+2	19/4
(2+3)/4+2	((2+3)/4)+2	9/4
(2+3)/(4+2)	(2+3)/(4+2)	5/6
2-3^2*-2+4	2-((3^2)*(-2))+4	24

In the last case, the True Basic compiler won't even accept the arithmetic statement because it has no idea what you really mean when you write * - . Contiguous arithmetic operators must be separated by parenthesis.

B) String Variables and String Constants

String constants are defined as any list of characters enclosed in quotation marks. Thus the phrases: "Beam me up Scotty" or "The velocity of the projectile was 2.78 km/sec" are strings.

We remarked earlier that True Basic was case insensitive to all types of variable names, keywords, but string constants are an exception. The phrases: "Now is the time" and "NOW is the time" are two different strings. This makes sense when one realizes that every character, i.e., letter, number or symbol, on the keyboard is assigned a numerical value in the ASCII (see the appendix of your text) table. For example, the return key is given the value 13, the ESC key is 27, 'A' is 65 while 'a' is 97, 'B' is 66 and 'b' is 98 and so on. When strings are compared for tests of equality, it is the ASCII value of each character that is used for that comparison. Unless the corresponding ASCII codes match exactly, the strings will not be equal.

Assigning a string constant to a string variable is identical to the procedure for assigning a numeric constant to a numeric variable ***with the exception that the name of the string variable must end in a dollar sign, $.*** For example:

String Variable → Time$ = "12 O'Clock" ← String Constant
ErrorMessage$ = "Function has a singularity at X = 0"
X$ = ""
DataString$ = "-1.2452"

In the case of X$ = "", the quotation marks without a space between them is defined as the 'null string'. Note that X$ = " " is not a null string but rather a string consisting of a single blank character. String variables are all initialized to the null string at runtime just as numeric variables are initialized to 0. Also notice in the last case that DataString$ is defined as a string consisting of digits, which without the quotation marks, would be an ordinary number. As we shall see, the language is rich with operators that can manipulate strings including extracting from strings like DataString$, their equivalent numerical value. The importance of strings and string operations cannot be overstated. Word processors, databases, indeed, most professional programs consist largely of string manipulation routines. Even most programs that do numerical computations on input from the keyboard do so by accepting numbers as string constants and not as numbers.

Finally, everything we have said about numeric variables and their values being associated with a location in computer memory apply equally to string variables and their values.

COMMENTS ON THE STORAGE OF NUMBERS AND STRINGS

Floating point or real numbers are numbers with decimals. Real numbers, integers and strings are all stored differently in memory. For example, real numbers are stored using 64 bits (also called double precision storage) or 8 bytes, in accordance with the IEEE format provided the computer has a math coprocessor. Otherwise a special internal format is used. On the other hand, integers are automatically stored using 16 bits. Strings use 8 bits (1 byte) per character.

The range of numeric input in True Basic is approximately -2e-308 to 2e+308 while the range of accuracy of any computation is 14 to 16 digits (10 to 16 for transcendental functions like log, sin etc.).

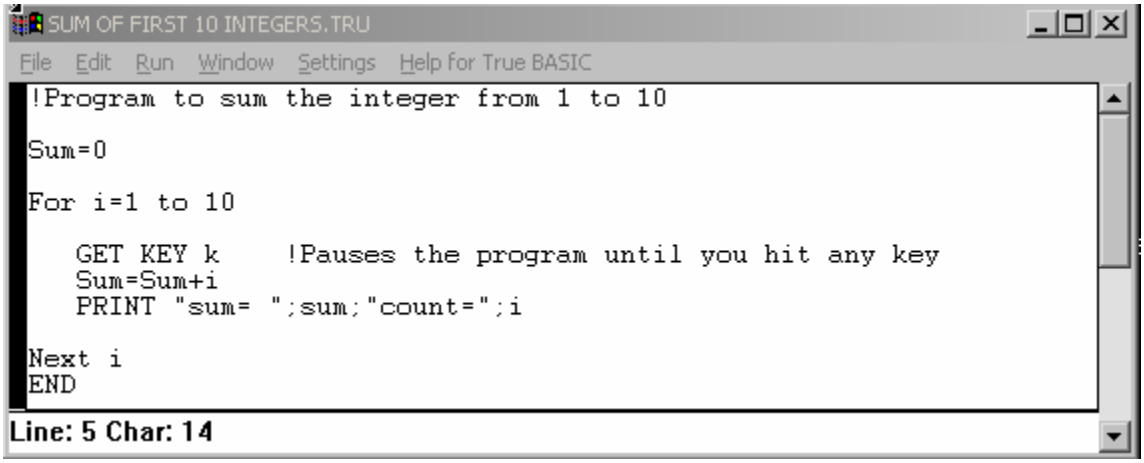
PROGRAMMING ERRORS (or Exceptions)

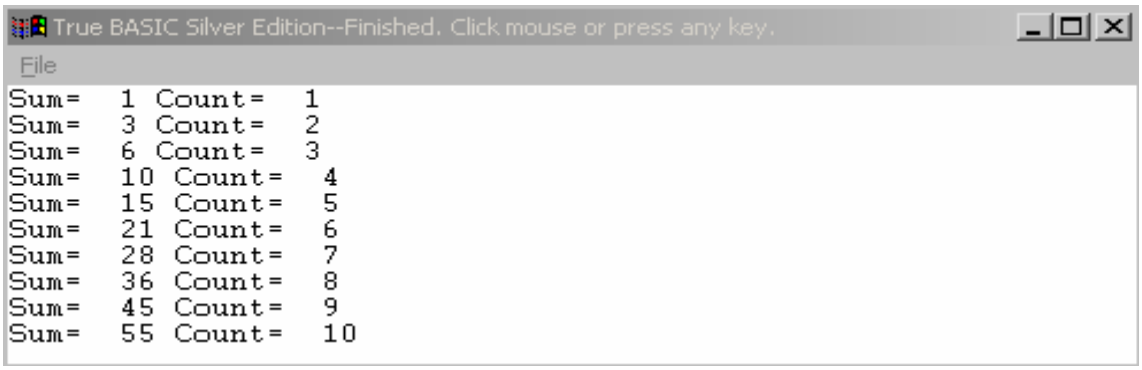
In the course of writing a program you will make many errors or, as they are often called, *exceptions.* Some of these errors are 'fatal' in that they result in an immediate halt of the program, either during the compilation process ('compile-time' errors) or at some point after the program finishes compilation ('runtime' errors). For 'compile-time' errors, the compiler will stop compilation immediately upon finding errors like misspellings or illegal statements or structures. These errors, which are relatively easy to fix, will result in an error message in the error window with a corresponding indication in the source window as to where the error occurred. Be warned however that the error the compiler tells you about may not be the problem at all, but rather the problem is to be found in earlier lines of code. It is just that the compiler accepted as correct an earlier statement and then found the incompatible code statement later. So when you look at the indicated error and you see nothing wrong, look at the earlier code for the problem. This problem is characteristic of all compilers irrespective of the computer language.

'Run-time' errors are usually accompanied by an error statement like: 'Division by Zero' or 'Channel isn't open'. Correction usually means re-writing the code to fix the problem. These problems are also trappable with an ***error handler*** to avoid a program crash. We will discuss the use of error handlers later.

ALGORITHMS

An algorithm is simply a program or program structure that solves a specific problem. For example, a recipe for making a chocolate cake is an algorithm as is the following code that sums up the integers from 1 to 10, the result of which is 55.





How would you alter the code to sum the first 1000 integers? Better yet, how about n integers where the user input the value of n?

This algorithm is a simple example of code that accomplishes exactly one thing, the computation of the sum of the first 10 integers. The Gas Law program is an example of a less well-defined algorithm. Most computer programs consist of many individual algorithms, and in the case of large programs, many thousands.

It is also worth noting that not all algorithms designed to compute the same quantity are created equal. For many algorithms, the issue of efficiency is not important, but for some, especially those involving data structures, it is crucial.

We now begin a systematic discussion of the set of statements, operators and structures that define the True Basic language. Examples will be used to illustrate programming concepts as they are needed. You should type in the code fragments, run them and then alter them to see what happens.

THE LANGUAGE
The following table contains a few of the most important statement that we will need along with some comments about their use.

STATEMENT	**COMMENT**
CLEAR	Clears the Output Window
REM or !	Remark statement – use the ! as it is more flexible
STOP	Stop program execution
END	Denotes the end of the program – ***this is a required statement***
PRINT	See below
TAB	Useful for making column data
PRINT USING	Used to format output
INPUT	See below
INPUT PROMPT	See below
LINE INPUT	See below
READ/DATA	Useful method for entering encoded data into memory

COMMENTS:

REM
REM is a non-executable (i.e., the compiler ignores it) statement that allows you to write explanatory comments to yourself and your fellow program users that will hopefully give you, and them, some idea as to why you wrote a section of code the way you did. The REM statement can only be used at the beginning of a line. A more useful alternative to the REM statement is the ! mark which can be used at the beginning of a line or at the end of a line of code to make comments or remarks. Another advantage of the ! mark is that if your comments extend over several lines you can ignore the ! mark until you are finished with your comments, then highlight the remarked section, press the key combination SHIFT ! and the entire section will be remarked out. This trick is also useful to temporarily remove sections of code, without having to delete them, so that they will not execute when you run the program.

Including comments in a program may seem a pretty obvious issue, but the importance of proper commenting cannot be over estimated. The fact is, most programmers, including professionals, do an entirely inadequate job of commenting their code and the reason is understandable. Commenting is boring, tedious and seemingly non-productive compared with actually writing code. The importance of carefully documenting your program becomes painfully obvious however when you try to understand the code a few months or even a few weeks later. And pity your poor colleague who, a year or more after you have moved on, is given the task of modifying your code. Without decent commenting, any hope of understanding what you did is all but impossible for a program of even modest size.

STOP
STOP does just that, it halts execution.

END
END is always the mandatory last statement in a program without external subroutines. It functions to halt program execution and gives a sense of closure to the code.

REM, !, END and CLEAR where all used in the Gas Law Program.

PRINT ZONES: True Basic, by default, divides the screen into columns of width16 characters each.

PRINT STATEMENT
The **PRINT** statement is used to direct output to the screen, or through the use of a directed channel, to a printer or to a file on a floppy or hard disk. The default option is to the screen and we will limit our discussion to this choice for the time being.

The PRINT statement can be followed by a TAB statement, any constant (string or numeric), any variable (string or numeric), any arithmetic statement or any combination of these. Multiple items to be printed are separated by commas or semi-colons, often called ‘delimiters’, and work as follows:

Commas: Commas indicate that each item to be printed should begin in a new print zone. In other words, the comma functions as a tab key on a typewriter where, in True Basic, by default, the tab stops are set at the beginning of each 16 character width column.

Semi - Colons: A semi-colon acts to eliminate spaces between printed items. In a sense, if we view the printing of characters on a screen like typing characters on a typewriter where a new line is generated by a carriage return, then the semi-colon acts to suppress that carriage return so the next character follows the last character on the same line. A PRINT statement by itself causes a blank line to be printed and for that reason is useful when one wants to separate lines of printed data.

EXERCISE 1 - Use of Delimiters: the comma and semicolon
The following short program illustrates the use of the comma and semi-colon when printing integers. In order to conveniently illustrate how the delimiters work we will briefly introduce a simple FOR/NEXT structure. The purpose of following code fragment is to again print the integers 1,2,3...,10. How the output looks will depend on what delimiter you use after the i in the PRINT i statement.

```
Sum=0
For i=1 to 10
      Sum = Sum+i
      PRINT "Sum= "; sum;
Next i
END
```

First, how does the program work? We begin by initializing Sum to 0 which is unnecessary (Why?) but it serves to remind you what the initial value is and I recommend you always initialize each variable. The **FOR/NEXT** structure functions to repeat whatever is between the **FOR** and the **NEXT** ten times; which, in this case, is the summing statement followed by the PRINT SUM statement. Thus our loop functions to print the first 10 integers.

Now type in the program, save it and then run it under the following conditions:

1) Without any delimiter after the PRINT i
2) With a comma after the PRINT i
3) With a semi-colon after the PRINT i
4) Replace PRINT i with the statement PRINT STR$(i).The statement STR$( ) functions to convert a numeric variable to a string so each of the integers, which were numbers, now become strings. Now re-run the program using commas after the PRINT i statement. This time there should be no spaces between the printed characters like there was between numbers. This is because strings are printed exactly as they are and not as formatted numbers.
6) Type in a PRINT statement just before the NEXT i. This illustrates the function of the PRINT statement as a spacer.

How Numbers are Formatted: We have just seen that numbers seem to be printed in a specific format. Like all computer languages, True Basic has specific conventions for printing numbers. These are,
1) all numbers end with a space and are confined to a single print zone
2) if a number is an integer, and can be expressed with 12 or fewer digits, it will be printed that way, otherwise it will be expressed using scientific notation where the largest number of characters is 15 including the - sign, the decimal point and any exponential notation if required. The number will be rounded to a maximum of 8 digits. This is consistent with a 16 character print zone. The same convention is used for floating point numbers. Thus the number -12345678901234567890e100 would be expressed as -1.2345679e+119 that is, to a total of 15 characters and rounded to 8 digits.

The following examples should help clarify these rules.

NUMBER	OUTPUT	COMMENTS
±123456789012	±123456789012	12 digit pos. integer, 14 characters.
±1234567890123	±1.2345679e+12	13 digit pos. integer, 15 characters
1234567.8	1234567.8	8 digit floating point no.
12345678.9	12345679.	9 digits rounded to 8
0.1234567890123	0.12345679	13 digits rounded to 8
-123456789.9	-1.2345679e+8	rounded + scientific notation
-12345678.9e-110	-1.2345679e-103	maximum of 15 characters

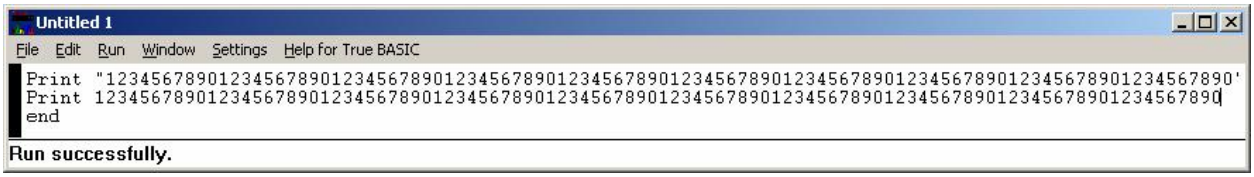
The purpose in limiting the numbers of digits displayed is to provide a convenient default format for tabulating data. Much greater flexibility in printing is obtained through the use of the **PRINT USING** statement which we will discuss shortly.

Printing Multiple Items on a Line

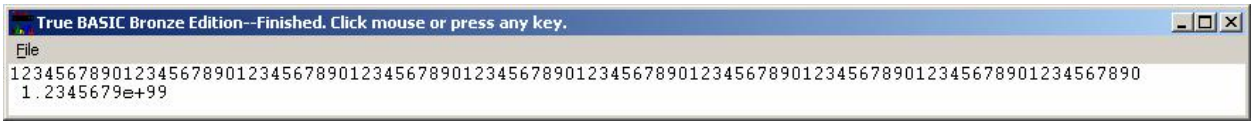
1) Constants and Numeric/String Variables: Statements like PRINT 2.2,-4,"1000" will be printed as specified, i.e., 2.2,-4 and 1000 where the item "1000" is a string, because of the quotes. The 2.2 will be printed in zone 1, -4 in zone 2 and the 1000 in zone 3. However, had semi-colons been used instead of commas, then each item would have been printed with a single space between them.

Printing strings differs from printing numbers as print zones are ignored for strings and there is no formatting as the following sceen shot shows:

THIS PROGRAM ILLUSTRATES HOW TB PRINTS STRINGS VS NUMBERS



OUTPUT



Note that the only difference between the two Print statements is that the first is a string (quotation marks) and the second is a number. Numbers are formatted and print in zones depending on the delimiter whereas strings print exactly as written.

Returning to the Gas Law Program we have the statements:

PRINT "Vol=";V

Here the string "Vol=" is printed first followed by the value of the numeric variable V, the two being separated by a semi-colon which ensures that the value of V will be printed immediately following the string "Vol=", as in, for example, Vol=50. This is a typical format for printing where one wishes to identify the variable name as well as its value. In the same way, multiple items may be printed on a single line as in,

PRINT "Pressure= ";P;"Temperature= ";T;"MolarVolume= ";V

2) Arithmetic Statements: As we have just seen, if a PRINT statement contains a numeric variable, then the current value of that variable will be printed when the PRINT statement is executed. As an alternative, we can also include as part of the PRINT statement the arithmetic statement that defines a particular numeric variable. After printing the string, the arithmetic statement will be executed and the result will be printed following the string. For example, in the next-to-last statement in the Gas Law Program, we have:

PRINT "Ratio of VDW to IDEAL Pressure=";ABS(VdwGasPress/IdealGasPress)

Here, following the string “Ratio of VDW to IDEAL Pressure=“ we have an arithmetic statement that calls for the absolute value of the ratio of the two gas pressures to be computed. Alternatively, we could have defined a new numeric variable called Ratio defined as

Ratio= ABS(VdwGasPress/IdealGasPress)

and then written:

PRINT “Ratio of VDW to IDEAL Pressure=“;Ratio

to accomplish the same end. From the view point of readable code, extensive use of arithmetic statements tied to PRINT statements is not advised.

TAB

TAB, when used in conjunction with PRINT, functions like it does on a typewriter. The snytax is:

TAB(*col*); moves the cursor to the column *col* on the line the cursor is already on, or

TAB(*row*,*col*); moves the cursor to the column *col* and the line *row*

NOTE: Tab cannot be used on its own but must be part of a print statement as in PRINT TAB (3,4).

For example,

```
V=125
PRINT TAB(10);”Vol = “;V
```

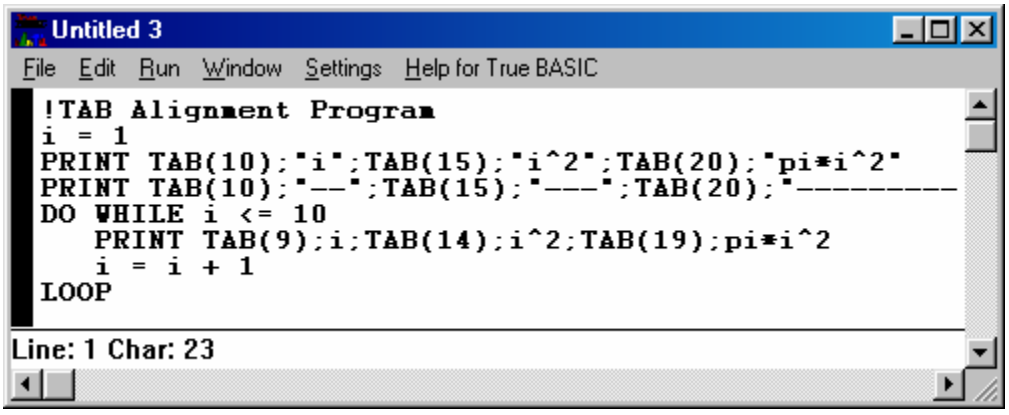
will move the cursor to column 10 on the present line and then print the statement Vol = 125.

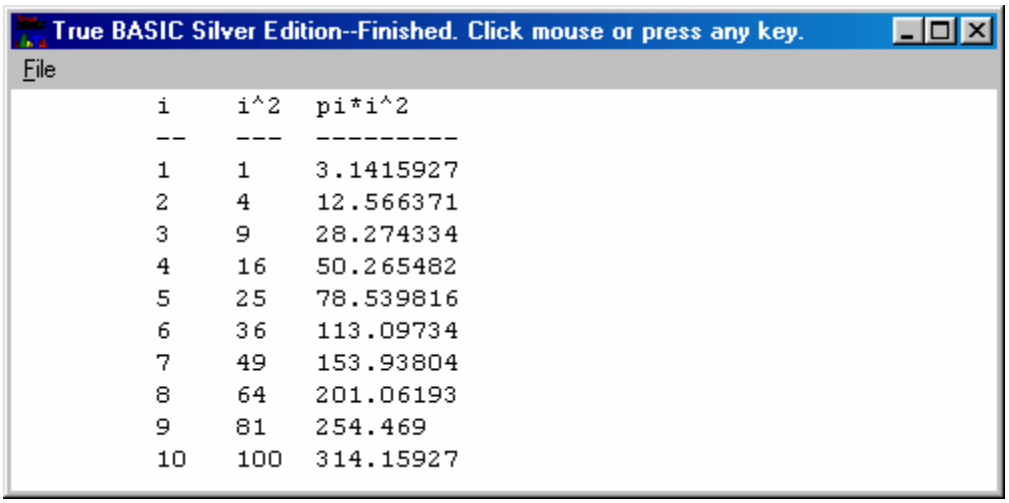
Multiple TAB statements can be used to override fixed zone widths. For example:

PRINT tab(10); x; tab(20); y; tab(30); z

will print the current values of the variables x,y and z beginning in columns 10, 20 and 30 respectively.

We can illustrate the TAB statement by modifying the previous program to read:





As you can see, the output lines up nicely beneath the different columns. Still, the output can be awkward when using the TAB function – in fact, if the output is negative, or the magnitude of i is negative then output will not look good. The **PRINT USING** statement will allow us greater flexibility in controlling how things look.

PRINT USING
Print Using (for numbers) is a very useful means of formatting output to the screen or printer. The syntax is:

PRINT USING "A$ Descriptor": constant, numeric variable or arithmetic statement

where:

1) the string A$ is optional. If included, it will be printed exactly as is.
2) the descriptor is a symbol that functions to define the output format. A table of descriptors is listed below. The quotes and colon are required.

PRINT USING DESCRIPTORS

DESCRIPTOR	MEANING	COMMENTS
#	Print leading zeros as spaces	Usual formatting of numerical output
%	Print leading zeros as zeros	
*	Print leading zeros as asterisks	
+	Print numbers with leading + or -	
-	Print numbers with either leading space or -.	Used for aligning decimal points when printing financial quantities
$	Print leading $ sign	
.	Fixes decimal point	
,	Prints commas	
^	Scientific notation	as in 2.34e-105
<	Left justify string	strings only
>	Right justify string	strings only

EXAMPLES of **Print Using** for Numeric Variables

Example 1: Consider the following program

1) PRINT USING "Pi to 20 significant figures is #.####################":Pi
2) PRINT USING "#.###############":Pi
3) OutputFormat$="#.###############"
4) PRINT USING OutputFormat$:Pi

(Line 1)
This line corresponds to the previous syntax where:
A$= “Pi to 20 significant figures is
Descriptor = #.####################”
and Pi is the True Basic function that returns the value of π

The output from this line of code is:

Pi to 20 significant figures is 3.1415926535897925000

Here the # symbols act as place holders for the digits thereby ensuring that 20 digits will be printed with the decimal fixed after the first digit. We note however that while 20 full digits are printed, only the first 15 are correct (the last 5 should read 32384 not 25000). This is because the value of Pi, like any floating point number, is stored in memory, using just 64 bits (which is called double precision). Since Pi has an infinite number of digits, then too accurately accomodate the number to its maximum precision would require infinite computer memory. This uncertainty beyond the 15 digit, which is a fact of life for all floating point number storage, is often call ‘computer roundoff error’. Roundoff error is not an issue of concern at this point, but it is an interesting topic that we will need to deal with later.

(Line 2)
This statement omits the string A$ and limits the output to 15 digits since the last five are meaningless.

(Lines 3 and 4)
These lines illustrate an alternative method of using the descriptor part of PRINT USING. It produces the same output as line 2 except that it allows the user to define the descriptor on a separate line so it could be used by multiple PRINT USING statements if desired - in short, it adds flexibility.

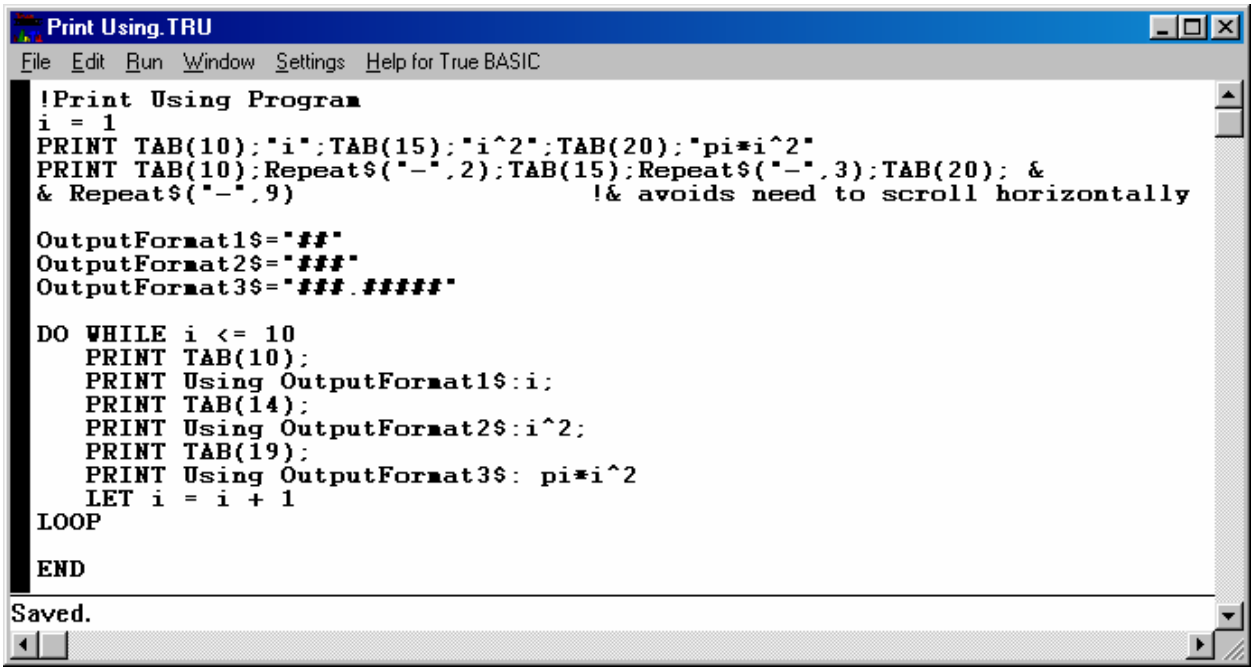
Example 2:
The following table illustrates the effect of various descriptors.

EXAMPLES OF PRINT/USING DESCRIPTORS (PU = Print Using)

DESCRIPTOR	OUTPUT	COMMENTS
PU “#.####”:1.234567890	1.2346	# acting as a place holder
PU “#.##########”:1.234567890	1.234567890	
PU “+#.####”:1.234567890	+1.2346	+ forces leading + or -
PU “%%%%.####”:1.234567890	0001.2346	% forces leading zeros
PU “#####.#^^^^^”:1.23456	12345.6e-04	scientific notation
PU “#####.#####”:12345.67890	12345.67890	
PU “#####.#####”:123456.7890	**********	string incompatible with format
PU “#####.#####”:1.234567890	1.23457	note rounding
PU “##,###,###,###”:12345678901	12,345,678,901	

Print Using is an extremely useful statement for formatting output. The most common descriptor for scientific output is the # along with the ^ for scientific notation. Business applications are more likely to use the $, comma, % etc.

You can also use the TAB function with PRINT USING but remember, ***they cannot appear on the same line***. A modification of the Tab Alignment Program illustrates the use of PRINT USING.



This modification produces nicely aligned output. Note the following:

1) the use of the Repeat$ function - see your text
2) that Print Tab and Print Using are on different lines – essential!
3) the way the do/loop is offset from the rest of the program and the code between the do and loop are further offset by another 3 (arbitrary) spaces. In addition, there is a blank line between the do/loop and the code above and below this structure. This is called white space. By indenting the code for spectific structures like the DO/LOOP and others that we will routinely use together with the liberal use of white space helps code readability and is strongly recommended.

You should copy the above modification, verify that it produces aligned output and then play with it. Try the other descriptors and see what happens.

INPUT STATEMENTS

This is the basic statement that transfers information from the keyboard to computer memory. In the Gas Law program the code:

```
PRINT “What is your volume in liters”;  !this is your prompt and what you see on the screen
INPUT V                                 !this pauses the program and waits your response
```

initiated the prompt for the volume from the user through the use of the PRINT statement. The INPUT statement is then executed resulting in a question mark being displayed on screen. The program then waits for a user repsonse. Typing a number and ***pressing return*** then transfers the data typed to the memory location associated with the variable indicated - e.g., V in the Gas Law program.

The input variables can be numeric or string or any combination of them, as in:

```
INPUT Pressure                      ! single numeric variable
INPUT X,Y,Z                         ! 3 numeric variables
INPUT ParagraphLength$              ! single string variable
INPUT x,y,z,x$,y$,z$,DataString$    ! mix of string and numeric types
```

Note that multiple variables are always separated by commas. When a multiple variable INPUT command is executed the program still responds with a ***single*** ”?”. In this case, the keyboard input must agree with both the number and type of variables called for and each datum must be separated by a comma. In the last case for example, the input: 1,2,3,4,5,6,7 would be perfectly alright because the number 1,2 and 3 would be assigned to the numeric variables x,y and z and the variables 4,5,6 and 7 would be treated as strings and assigned to x$,y$,z$ and DataString$. On the other, the statement INPUT X,Y,Z calls for three numeric variables so the input: 1, Klingon, 3 would result in an error message and the request for you to reenter the data because Klingon is a string variable, not a numeric variable.

Comment: Errors made when responding to the INPUT statement are not fatal in that they do not crash the program. Instead, an error message appears and waits for you to respond correctly. An alternative and much preferred way to input data is to not use the INPUT statement at all, but instead use the GET KEY statement in combination with the CASE SELECT structure. With this code combination, one can trap errors of the wrong data type before they are even seen on screen. We shall use this technique later after we have introduced Decision Structures.

INPUT PROMPT
This is a useful alternative to the two line combination of PRINT and INPUT used above. Here the prompt is part of the input statement as in:

INPUT PROMPT "some message":var1,var2,...

where "some message" is the prompt and var1, var2 etc. are the variables to be assigned the input constants. Thus the previous PRINT/INPUT combination would read:

INPUT PROMPT "Enter your volume in liters": V

One advantage of this structure is that it does not result in a '?' being printed as it does everytime an INPUT statement is executed.

LINE INPUT
LINE INPUT is used for string variables. The syntax is :

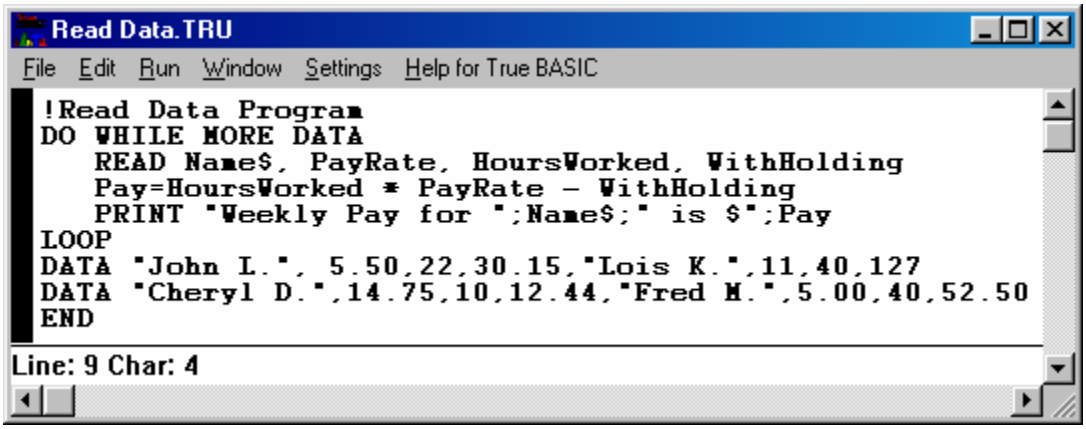
LINE INPUT *var1*$, *var2*$
or
LINE INPUT PROMPT *prompt*$: *var1*$, *var2*$

READ/DATA
The READ/DATA statement is used to input data to memory from within a program rather than from the keyboard. The syntax is:

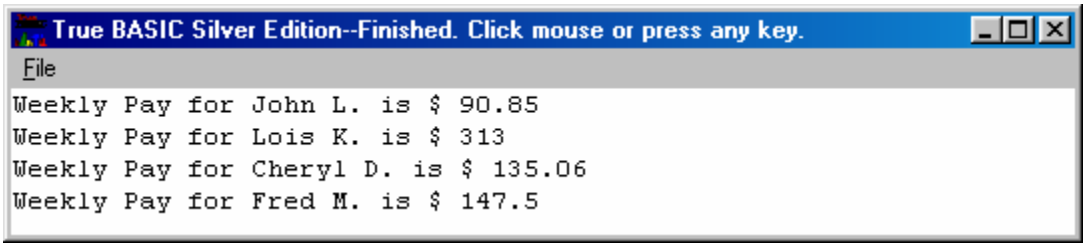
READ var1,var2,.....
  code block
DATA
RESTORE (optional)

The READ statement simply lists the variables to which you wish to assign the constants, both numeric and string, specified in the DATA statement. The RESTORE statement is optional.

EXAMPLE:
Here is a short program to compute the pay packet for some part-time workers.



Here's the ouput:



The Way Read /Data Works

The READ statement lists the variables, one string and three numeric, needed to compute the current value of the Pay variable. During the first do/loop pass, the string constant "John L." is assigned to the string variable Name$, and the next three numeric constants to the numeric variables PayRate, HoursWorked and WithHolding. Execution then drops to the Pay=.. statement and then to the PRINT statement. Program flow then loops back when LOOP is executed and the next four items in the DATA statement are assigned. This is done (in this case four times) until the the number of items in the DATA statement has been exhausted.

Clearly then, the number of items in the DATA statement must be a multiple of the number of variables to be assigned. In addition, the items in the DATA statement must agree in kind with the variables to which they are assigned - in short, if the third variable to be assigned is a string variable then the third item in the DATA statement must be a string constant and not a number. Otherwise you will have a fatal error (Data item isn't a number or Reading past end of data)!

Now re-run it with the WHILE MORE DATA part of the do/loop commented out (use the !). Now you should get the error message 'Reading past end of data'. That is the function of the WHILE MORE DATA (or alternatively, UNTIL END DATA) is to halt the looping process when the data has run out thus avoiding the error message.

As the data is read, an imaginary pointer moves from data item to data item until the last constant is read and assigned (in this case 52.50). The pointer then moves to the next item which is non-existent. It is this final positioning of the pointer that causes two problems: i) the "Reading past end of data" error and ii) no way to re-run the program and use the same data without having to stop the program and re-run from scratch. The solution to the first problem is to use the DO WHILE MORE DATA statement and the solution to the second is to include, after the LOOP statement, a RESTORE statement which has the affect of moving the pointer back to the first data item, namely, "John L."

On the next page is a program that summarizes the various Input/Output methods as well how to format output. The output from the program follows.

Inputout.tru

File Edit Run Window Settings Help for True BASIC

```
! ----------------------DATA INPUT/OUTPUT METHODS-------------------------
! HERE WE USE THE INPUT, READ/DATA, TAB ETC, STATEMENTS TO ILLUSTRATE HOW
! TO ENTER AND PRINT DATA.  THE ALGORITHM CONVERTS DEG.'S C TO DEG.'S F.

PRINT "*****************************INPUT************************************"
     PRINT "Enter two Centigrade temperatures"
     INPUT C1,C2           !note the comma
     LET F1=(9/5)*C1+32    !here is the algorithm for F1
     LET F2=(9/5)*C2+32    !likewise for F2
     PRINT "Your Centigrade temperatures are: ";"C1=";C1;"AND C2=";C2
     PRINT "Your Fahrenheit temperatures are: ";"F1=";F1;"AND F2=";F2
PRINT"*****************************READ/DATA********************************"
READ C1,C2  !the READ is ignored until the DATA is found, and then processed
     LET F1=(9/5)*C1+32
     LET F2=(9/5)*C2+32
     PRINT "Your Centigrade temperatures are: ";"C1=";C1;"AND C2=";C2
     PRINT "Your Fahrenheit temperatures are: ";"F1=";F1;"AND F2=";F2
DATA 37,102
RESTORE !not needed here, but this is where it would go if it were.
PRINT"*****************************TAB**************************************"
READ C1,C2
   LET F1=(9/5)*C1+32
   LET F2=(9/5)*C2+32
   PRINT "Your Centigrade temperatures are: ";"C1=";C1;"C2=";C2
   PRINT !spacer
   PRINT TAB(10);"F1"; TAB(20);"F2"
   PRINT TAB(8);F1; TAB(18);F2
DATA 37,102
PRINT"*************************PRINT/USING********************************"
READ C1,C2
   LET F1=(9/5)*C1+32
   LET F2=(9/5)*C2+32
   PRINT "Your Centigrade temperatures are: ";"C1=";C1;"AND C2=";C2
   PRINT !spacer
   PRINT TAB(10);"F1"; TAB(20);"F2"
   PRINT TAB(8);REPEAT$("-",6);tab(17);REPEAT$("-",7) !repeat$ is a string fn.
      PRINT TAB(7);                  !note:  print using and tab
      PRINT USING "###.###":F1;      !cannot be on the same line.
      PRINT TAB(17);
      PRINT USING "###.###":F2;
DATA 37,102
END
```

Saved.

True BASIC Silver Edition--Finished. Click mouse or press any key.

File

```
*****************************INPUT************************************
Enter two Centigrade temperatures
? 37,102
Your Centigrade temperatures are: C1= 37 AND C2= 102
Your Fahrenheit temperatures are: F1= 98.6 AND F2= 215.6
*****************************READ/DATA********************************
Your Centigrade temperatures are: C1= 37 AND C2= 102
Your Fahrenheit temperatures are: F1= 98.6 AND F2= 215.6
*****************************TAB**************************************
Your Centigrade temperatures are: C1= 37 C2= 102

         F1         F2
       98.6       215.6
*************************PRINT/USING********************************
Your Centigrade temperatures are: C1= 37 AND C2= 102

         F1         F2
      ------    -------
      98.600    215.600
```

LIBRARY FUNCTIONS

Our programming examples so far have used only a very limited number of possible True Basic statements. In order to expand our choices we now introduce a few of the more important True Basic library functions. Most of these are for numeric functions since we are leaving string operations until later. See the True Basic Reference manual for a complete listing. We will detail the built in functions for strings later. Each of the listed functions is called with the statement

Numeric or String variable = Library Function(Variable Name)

as in, Y=SQR(X) where X is some positive number already in computer in memory.

PARTIAL LIST OF TRUE BASIC LIBRARY FUNCTIONS

FUNCTION	NAME	COMMENTS
ABS (x)	Absolute Value	
SQR(x)	Square Root	x>=0
EXP(x)	Exponential	exception 1003 - overflow
LOG(x)	Log base e	x>0, exception 3004 - LOG of number <0
LOG2(x)	Log base 2	ditto
LOG10(x)	Log base 10	ditto
SIN(x)	Sine of x	x in radians by default
COS(x)	Cos of x	ditto
TAN(x)	Tan of x	ditto
INT(x)	Integer - see below	
CEIL(x)	Ceiling - see below	Defined as -INT(-x)
RND	Random number	returns a psuedo random x where 0<=x<1
ROUND(x,n)	x rounded to n places	see reference manual
MOD(x,y)	Modulus - see below	exception 3006, y cannot be 0
VAL(x$)	Value of string x$	exceptions 1004,4001 -see ref. manual
STR$(x)	String operator	converts x to a string truncated to 8 digits
CHR$(x)	Character string	0<=x<=255 where x is an ASCII code
ORD(a$)		returns ASCII code of a$

COMMENTS and EXAMPLES

Most of these functions are pretty obvious. Some need additional explanation however, including:

INT(x): This function returns the greatest integer <= to x. Thus INT(2.99)=2 , INT(-2.01) = -3 and INT(25) = 25. INT is sometimes called the FLOOR function.

CEIL(x): Returns the least integer that is >=x and can be defined as -INT(-x)

RND: The statement, x = RND will generate a single random number such that 0<=x<1. Note that x <>1. If a sequence of random x are generated using a DO/LOOP for example, the sequence will be found to be repeatable from run to run. This makes debugging possible. To remove this repeatability, include the RANDOMIZE statement. To generate random numbers in the range between some upper bound U and some lower bound L, use either:

N = INT[(U-L+1) * RND+L] for random integers between U and L

N = ROUND[((U-L) * RND+L),J] for random 'real' numbers between U and L

where J is the number of significant figures wanted.

For example, suppose we want to generate the random integers 0 and 1 in order to represent a head or tail when a coin is flipped. We take U=1 and L=0 so the statement: N = INT(2*RND) will simulate a coin flip quite nicely since values of RND from 0 to 0.4999... produce a 0 while values of RND from 0.5 to .999... produce a 1, where both ranges are equally probable.

Excercise: Write a program to simulate the rolling of a die. Keep track of the number of times each face comes up for say, 10,000 rolls. Are your results reasonable?

MOD: MOD(x,y) returns x modulo y provided y<>0. It is formally defined as:

$$MOD(x,y) = x - y * INT(x/y)$$

and, provided that x and y have the same signs, computes the remainder when x is divided by y. Thus:

MOD(5,3) = 2
MOD(5,-3) = -1 i.e., 5 - (-3) * INT(5/-3) = 5 - (-3) * INT(-1.666...) = 5+3* (-2) = -1
MOD(-5,-3) = -2
MOD(5.3,7.9) = 5.3 since 5.3/7.9 = 0 with remainder = 5.3
MOD(7.9,5.3) = 2.6

Mod is very useful at times. For example:

1) INTEGER TEST: MOD(x,1) = 0 if x is a positive integer, and equals the fractional part of a positive x if x in not an integer. This is a good way to extract the fractional part of a positive number. Try a negative number like MOD(-1.23,1) to discover the rule for extracting the fractional part when x<0).

2) TEST FOR INTEGER PARITY: MOD(x,2) =0 if x is an even integer and non-zero if x is odd. This test is equivalent to the comparison: IF x/2 = INT(x/2) THEN etc. Do you see why?

VAL(x$): If x$ is a string capable of being converted to a number then VAL will return the numerical equivalent of x$ as in:

VAL("105.345") = 105.345 where the result is a numeric constant, not a string. VAL("1,111,111") will cause an exception (commas are not allowed for numbers). VAL("I love programming") will also cause an exception because VAL cannot extract a value.

The VAL function is very useful because properly written programs will input data as strings and not numbers. However, at some point one needs to use these strings as real numbers and that is where the VAL function is used - it extracts the number without altering the string.

STR$(x): Here we convert a numeric variable to a string, as in STR$(123.456), and return the string equivalent, "123.456". This is another extremely useful function.

In addition to the above table of functions, there are a number of other important and useful commands that we will have occasion to use. These will be usually introduced in code examples, and will be discussed at that time.

DECISION STRUCTURES

As we have seen, computer programs are executed serially, that is, one line after the other until there is a statement that shifts program flow to another part of the program. There has to be some method to logically re-direct the flow when needed and that method makes use of decision structures like **IF/THEN/ELSE** and **CASE SELECT**. However, in order to take advantage of these structures we need to first know something about both *relational operators* and *logical operators*.

RELATIONAL OPERATORS

The relational operators are the following,

RELATIONAL OPERATORS

Operator	Meaning
=	equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
<>	not equal to

Here, the = sign actually functions as an equals sign and not as a replacement or substitution operator as has been the case until now.

LOGICAL EXPRESSIONS
Logical expressions result from the use of the relational operators to compare numeric or string variables, constants or expressions. The result of the comparision is either 'true' or 'false'. Thus, if x=6, then the statement x>4 is 'true' while the statement x<=4 is false. String comparisons are based on the numerical values of each string character in the ASCII table so a statement like "I">"i" is equivalent to asking of the respective ASCII values if 73>105 (obviously false). More complex logical expressions can include multiple tests using the relational operators in conjunction with one or more *logical operators*.

LOGICAL OPERATORS
There are three *logical operators* (also called Boolean operators) used in True Basic: **OR, AND,** and **NOT**. These operators are then used together with the relational operators to form more complicated logical expressions.

Examples of logical expressions include,

$$\textbf{IF}\ \left\{\begin{array}{l}\textbf{x>0 AND y<>x}\\ \textbf{x\^2 + y\^2 < r\^2}\\ \textbf{A\$ <> B\$ OR (A\$ \& B\$ <> ""AND Ans\$ = "Y")}\end{array}\right\}\ \textbf{THEN do something}$$

When evaluated, these expressions are either 'true' or 'false'. In the first expression, for example, only if x really is greater than zero and only if x and y are not equal will the expression be found 'true', otherwise the expression is 'false'. The second example is a case of a logical expression without a Boolean operator. In the last expression, the '&' is called a *concatenation* operator and functions like a + sign for strings. Thus if A$ = "abc" and B$ = "def" then A$ & B$ = "abcdef". The successive quotes "" is defined as the null string so the statement A$ & B$ <> "" will be 'true', if at least one of the concatenated strings, A$ and B$, is not null.

This last example illustrates that the interpretation of a logical expression can get complicated.

The following *truth table* shows how the logical operators work in complex logical expressions like A$ <> B$ **OR** (A$ & B$ <> ""**AND** Ans$ = "Y"). LE1 is logical expression 1 (like A$ <> B$) and LE2 is logical expression 2 (like A$ & B$ <> ""**AND** Ans$ = "Y").

TRUTH TABLE FOR THE LOGICAL OPERATORS OR, AND, NOT

LE1	LE2	LE1 OR LE2	LE1 AND LE2	NOT LE1
true	true	true	true	false
true	false	true	false	false
false	true	true	false	true
false	false	false	false	true

Thus, for example, if LE1 is true and LE2 is false, then when connected by an **OR**, the result is still true, but if the operator is **AND**, the result will be false. In other words, the **OR** operator requires that only one of the two logical expressions be 'true' for the combined expression to be 'true' whereas the **AND** operator requires that both be 'true'. The **NOT** operator simply reverses the state of the expression from 'true' to 'false' and conversely.

Parantheses should be used in complicated logical expressions to control the intended logic. For example, the statement

(A>0 **AND** B>0) **OR** (C>0 **AND** D>0)

is 'true' when A and B are both >0, or C and D are both >0, or when all four are >0. However

A>0 **AND** (B>0 **OR** C>0) **AND** D>0

is true when A and D are both >0, and either B or C are >0.

The logical expression,

A$ <> B$ **OR** (A$ & B$ <> “”**AND** Ans$ = “Y”)

is ‘true’ if the two strings A$ and B$ are not equal, or if the expession in parentheses is true, which requires that the concantenated strings A$ and B$ not equal the null string “”, and in addition, that the string Ans$ equal “Y”.

We have not said much about the **NOT** operator and that is because we don’t recommend its use. **NOT** usually just obfuscates the code and you are usually better off changing the logic to avoid it.

Operator Hierarchy
We have already seen that there is a execution hierarchy for arithmetic operators, and the same holds true for the relational operators and the logical operators. The following table shows the precedence of execution for all three types of operators.

ARITHMETIC, RELATIONAL AND LOGICAL OPERATOR PRECEDENCE

OPERATOR TYPE	OPERATOR	PRECEDENCE
ARITHMETIC	^ * , / + , -	Executed first
RELATIONAL	= , <> , < , <= , => , >	
LOGICAL	NOT AND OR	Executed last

It should be remembered that expressions contained in parentheses are always evaluated first.

We are now ready to explore the two decision structures available in True Basic.

a) **IF/THEN/ELSE**
This structure is one of the most important general branching tools available. There are two flavors available, single line and multiple lines. Note: here *le* denotes *logical expression*, and *tb* denotes True Basic)

1) single line syntax: IF *le* THEN *tb statement* ELSE *tb statement* Here the THEN part executes only if le is true otherwise, the ELSE executes.

The ELSE *tb statement* part is optional. If used, it functions as a catch all for what is not covered by the *le*. For a complete listing of acceptable *tb statement*s, see the True Basic Reference manual. Most statements that are part of the True Basic language will work, but if in doubt, try it and see. The compiler will happily let you know if you have guessed wrong.

The following code fragment illustrates how the single line structure works:

```
PRINT “Input two numbers and press return”
INPUT x,y
IF x>y THEN PRINT “x is > than y” ELSE PRINT “x is not > than y”
```

After the program assigns the two typed in numbers to x and y, execution passes to the IF expression where the truth of the statement x>y is tested. If x is greater than y, then the logical expression x>y is true so the PRINT statement following the THEN is executed. If x>y is false because x is actually less than y, then control passes to the ELSE statement and its PRINT statement is executed. Execution then passes to the next line of code. If the ELSE statement is omitted, then, if x>y is false, execution passes directly to the line after the IF/THEN. It is important to remember that the ELSE statement will always be executed if the IF expression is false because it covers all other contingencies.

Question: What happens if x = y ? The ELSE is executed which gives the ambigous answer that "x is not > than y" and illustrates that the single line construct is inadequate if there are more than two possibilites.

2) multiple line syntax:

```
IF le THEN
....  code block 1
  ELSE IF le THEN
....  code block 2
≈
  ELSE
....  final code block
END IF
```

The ≈ denotes that the structure can have any number of ELSE IF statements. Each code block can contain anywhere from a single statement to many statements. Again, the ELSE is not required, but note that the END IF statement is because it tells the compiler that the structure is closed or finished.

The way the multi-line structure works is similar to the single line case. Each IF or ELSE IF is tested sequentially until one is found to be 'true'. The TB statements contained within that particular block are then executed after which program flow drops to the line of code following the END IF. If none of the tests are 'true', control passes to the ELSE and its block of code is executed. Finally, control passes to the line following the END IF. Remember, ELSE is always executed if the prior logical expression is not. The multi-line option is the following modification of the previous example to take into account the possibility that x and y are equal.

```
PRINT "Input two numbers and press return"
INPUT x,y

IF x<y THEN
      PRINT "x is less than y"
  ELSE IF x>y THEN
      PRINT "x is greater than y"
  ELSE
      PRINT "x equals y"
END IF
```

Note that we could have used an additional ELSE IF for the x = y case and then either included or omitted the ELSE. If included, it would have been immediately followed by the END IF and therefore been non-functional.

Also observe how the use of indenting has improved readability.

The **IF/THEN/ELSE** structure can be nested many layers deep but if you find yourself using more than a couple of layers, re-write the code. For example, here is a program to determine the largest of three integers, x, y or z.

```
PRINT "Enter 3 integers"
INPUT x,y,z

IF x>y THEN            ! executed only if x>y

  IF x>z THEN              ! x largest
      Largest = x
    ELSE                   ! z must be largest as x<z and x>y
      Largest = z
  END IF

ELSE IF z>y THEN   ! y>x and z>y
      Largest = z
ELSE                  ! x<y and z<y
```

```
        Largest = y
    END IF

    PRINT "The largest integer is ";Largest
    END
```

You should type the program in, run it and make sure you understand how it works. You will find that as simple as it is, it does force you to think your way through the logic. This is certainly not the best way to find the maximum of three numbers but it does illustrate the logical difficulties of deeply nested decision structures. They become opaque very quickly. It also illustrates the importance of indenting to improve readability.

b**) SELECT/CASE**

Like **IF/THEN**, **SELECT/CASE** allows multi-way branching depending on the value of a parameter. The syntax is:

```
SELECT CASE (numeric or string variable or expression)
    CASE test 1
        Code Block 1
        .......
    CASE test 2
        Code Block 2
        .......
        .......
    CASE ELSE
        Final Code Block
END SELECT
```

The **SELECT CASE** statement may be followed by a string or numeric variable or expression like S$, x, GasPressure or x^2 * cos(x). The 'test' that follows the CASE statement must be a numeric or string constant **and not a variable**. For example, the following are acceptable choices,

```
CASE 5
CASE 1,2,3
CASE -1000 to 0, 1 to 1000
CASE "Dream on, Klingon"
CASE "A" to "Z"
CASE IS <> 6        !won't work
CASE IS < 0         !won't work
```

where in the last two cases, we have use the relational operators. The 'IS' modifier is required whenever a relational operator is used, whereas the statement,
CASE x >0 is ***not*** acceptable syntax because the *test* must be a numeric or string constant and not a variable. The logical operators AND, NOT and OR are also **not** allowed. Here is an example.

```
DO
    PRINT "Input an integer between 0 and 10"
    INPUT x
        SELECT CASE x
            CASE  0
                PRINT "x = 0 and is not prime"
            CASE 1
                PRINT "x = 1 and is not prime"
            CASE 2
                PRINT "x = 2 and is the only even prime"
            CASE 3,5,7
                PRINT "x is prime"
            CASE 4,6,8 to 10
                PRINT "x is composite"
            CASE ELSE
                PRINT "x is either non-integer or not between 0 and 10"
```

```
        END SELECT
LOOP
```

This program asks for you to enter an integer between 0 and 10 which is inputted to memory with the INPUT statement. SELECT CASE then takes x and tests it via the CASE statements until a 'true' statement is found in which case that code block is executed. Note the CASE ELSE catch all. The similarity between this structure and the IF/THEN/ELSE structure should be obvious.

The CASE ELSE is not required but an exception or error (Exception: 10004 No CASE selected, but no CASE ELSE) will occur if your CASE statements do not include every conceivable contigency and you do not have a CASE ELSE statement. Execution is then left high and dry with no place to go. To be on the safe side, always include the CASE ELSE statement.

SELECT/CASE is especially useful when constructing menus in a program as in the following example.

EXAMPLE: A Fail Safe Menu
Suppose you were writing a program that would have some basic file handling capability, that is, the ability to either open an existing file or create a new one. You would need a menu to prompt the user for these two options, and, in addition, give the user the choice of leaving the menu if they changed their mind.

In addition to providing the menu choices to the user, it is imperative that the menu be 'fail-safe' in the sense that the routine won't crash if the wrong key is pressed and that is where the CASE/SELECT construct becomes important. We have choosen to use the ASCII code to represent each choice including the function key F1 in case HELP is needed. There is no code following CASE statements because we are interested only in the construction of the menu here.

If you run the program, you will get a beep any time you press a wrong key (i.e., any key other than h, H, O, o, C, c, or Esc) and the cursor will stay fixed in one position until you get it right. You could have converted the ASCII code into a string using the CHR$ together with the UCASE$ (or LCASE$) commands so your CASE statements would then be written in terms of the ters 'O' and 'C' etc. However, ESC and the function key F1 do not have simple character representations so you would still have to deal with these keys in terms of their ASCII codes, so it makes sense to keep it simple.

There are two other points that deal more with esthetics than functionality. These are the CURSOR ON and OFF statements and the positioning of the DO statement below the menu itself. In professional programs you do not see the cursor off in some corner flashing away like some adventitious voyeur when it is not needed (in fact, in *Windows* programs, you rarely see the cursor except when numerical input or a pathname is required). So we turn it 'on' only to act as a prompt for the menu statement "Selection ?", otherwise it is 'off'.

```
Call GetDataMenu    !call statement
End

Sub GetDataMenu
DO                                  ! the DO goes here NOT above the menu
    SET Cursor "on"                 ! cursor should be 'off' when not needed - but it is here
    GET Key k                       ! fetches the ASCII code of the key pressed
    SET Cursor "off"
    SELECT CASE k                   ! k = ASCII code of pressed key
     CASE 72,104,315                ! H,h,F1 -  use any of them for help
       !code that calls a 'help' file
     CASE 27                        ! Esc
       EXIT SUB                     ! exits to call routine above
     CASE 79,111                    ! CASE O,o - Open a file
       lots of code the last
       statement of which is EXIT SUB
     CASE 67,99                     ! CASE C,c - Ccreate data file
       another whole bunch of code
       with an EXIT SUB
     CASE ELSE                      ! any other key pressed
       Sound 1000,0.1               ! beeps
```

```
      SET CURSOR LastRow,LastCol      ! reset cursor following ? mark
   END SELECT
  LOOP                                ! loop back without re-printing menu
END SUB
```

You can run the code frag Menu.tru to see how this works.

The first few lines are just the 'call' routine. While we have not covered subroutines at this point, it should be pretty obvious how the code works. The CALL GetDataMenu statement simply passes execution to the subroutine of the same name. When Esc is pressed, the subroutine is exited to the line following the line after the CALL GetDataMenu statement.

We also have some new commands which are shadowed in the code below. Let's look at the new commands first.

GET KEY x
This statement (where x is any numeric variable), serves to pause execution of a running program. Execution resumes as soon as any (almost 'any' that is, ALT and CTRL are exceptions) key is pressed. Once a key (not keys - GET KEY responds to a single key stroke), is pressed, the ASCII value of that key is then assigned to x, and furthermore, this happens instantly (or at least it seems like it) without the need for pressing the ENTER or RETURN key. For example, if the key Y is pressed, then the value 89 (the ASCII value of cap Y) is assigned to x. This value could then be tested for as part of an IF/THEN or a CASE/SELECT structure. This makes the GET KEY and CASE/SELECT expecially useful as part of a message prompt in a menu system.

SET CURSOR s$/ASK CURSOR s$
Here, s$ has two values, "on" and "off". With SET CURSOR "off" (or "on") you can toggle the cursor off or on so it does not interfere with how the screen looks when, for example printing a graphic. If there is uncertainity about the state of the cursor when running a portion of a program, one can have the program interrogate the state of the cursor and return s$ as either "on" or "off". s$ can then be tested with an IF/THEN and the cursor state changed if necessary. There are a whole series of ASK statements available in True Basic. See your text.

SET CURSOR(row,col)/ASK CURSOR(row,col)
Here SET CURSOR does just that, it moves the cursor to the requested location and waits for the next command. ASK CURSOR functions to find out the present position of the cursor so that you can have the code change it if you need to.

SOUND (freq,duration)
This key word produces a sound of frequency *freq* (in Hertz) for the time period *duration* (in seconds). Program execution continues while the sound is produced.

LOOP STRUCTURES

True Basic supports two kind of loop structures, the **FOR/NEXT** construct and the **DO/LOOP** that we have already used. Loop structures cause a program to execute a code block repeatedly. The distinction in the use of the **FOR/NEXT** and the **DO/LOOP** is that the **FOR/NEXT** requires that you know in advance how many times you want to repeat the loop, while the **DO/LOOP** is open ended.

a**) FOR/NEXT**
The syntax is:

FOR *numeric variable = initial index* **TO** *final index* **STEP** *increment*
    Code to be executed…
NEXT numeric variable

where the *numeric variable* can be any valid name for a numeric variable (i,j and k are typically used, but a more meaningful name is recommended), but not an expression, whereas *initial index*, *final index* and *increment* can be either a number or expression. The *increment* is optional with a default value of 1.

Here's an example:

```
FOR i = 1 to 10
      PRINT i,i^2
NEXT i
```

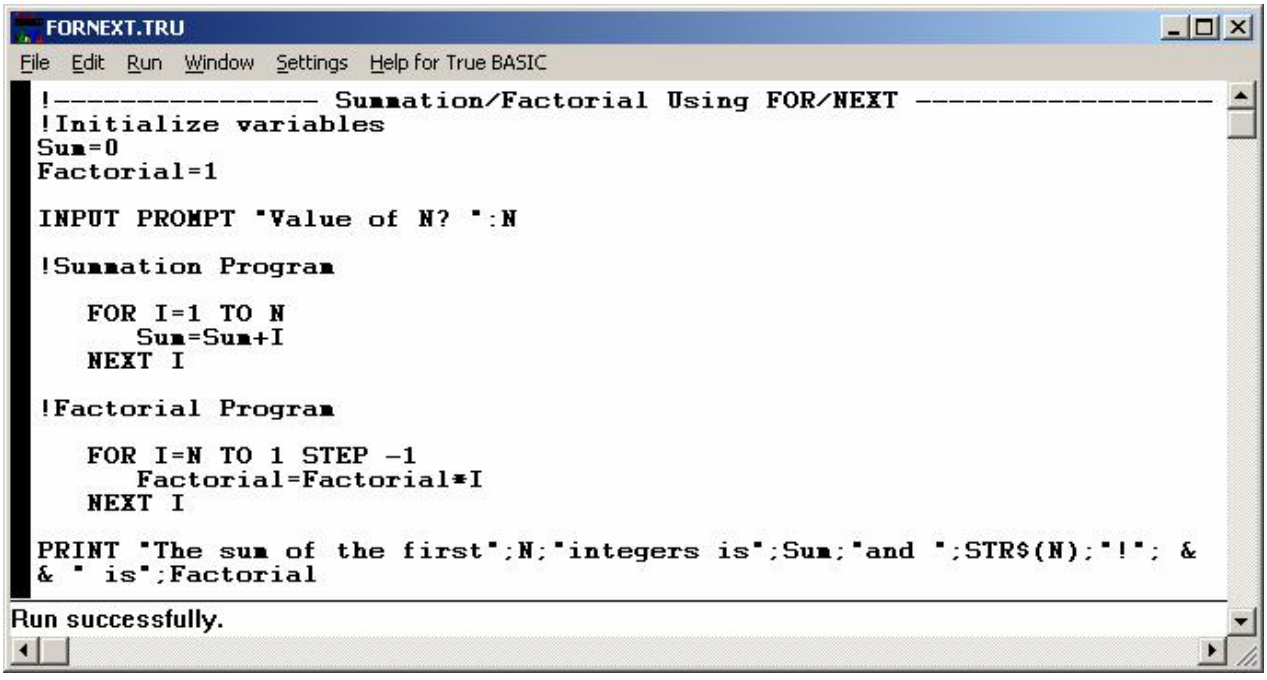
Running this fragment produces two vertical columns (print zones 1 and 2) of numbers, the first is the set of integers from 1 to 10 and the second, the squares of those integers. On the first time through, i is set equal to 1, the PRINT statement is executed and then the NEXT i increments i to i+1 = 2 and the PRINT again executed etc. The loop is executed until i = 11, that is, until i takes on the first value inconsistent with the final index. It is important to remember that i will always be one more than the last value executed. One can also exit the loop with a **EXIT FOR** statement as in:

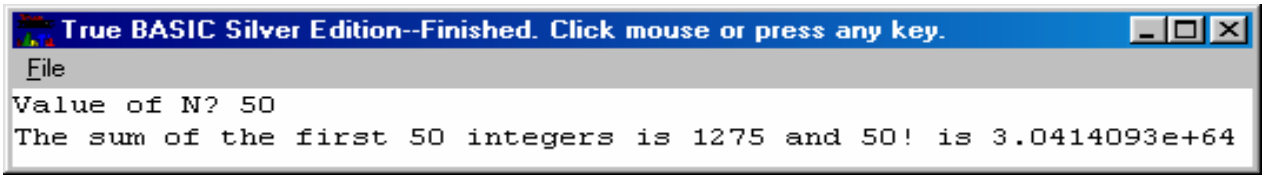
```
x = 100

FOR i = x TO 2*x^2 STEP x
      IF i > 10000 THEN EXIT FOR
      PRINT i;
NEXT i

END
```

This program prints the numbers from 100 to 10000 in increments of 100. Without the EXIT FOR, the program would print numbers through 20000 since when x=100, the final index 2*x^2 is 20000. Note that the EXIT FOR drops execution to the line following the NEXT i, which in this case happens to be an END statement.

Finally, note that neither the indices nor the increment absolutely have to be integers but the difference between the indices should be divisible by the increment if you want to end the loop with the loop variable taking on the value of the final index. Non-integer indices are avoided in general.

Our next program computes the sum and factorial of an arbitrary number of integers. Unlike a calculator, computer languages typically do not include the factorial as a library function so you have to write your own. The program listing follows together with a typical run for N=50.





ANALYSIS:
The summation part is very straight forward. i ranges from 1 to the input value of N. When i=1, Sum will equal 1, when i=2, Sum will equal 1+1 or 2, when i=3, Sum=2+1 or 3, ie, Sum is a running total whose final value will be 1+2+...+N.

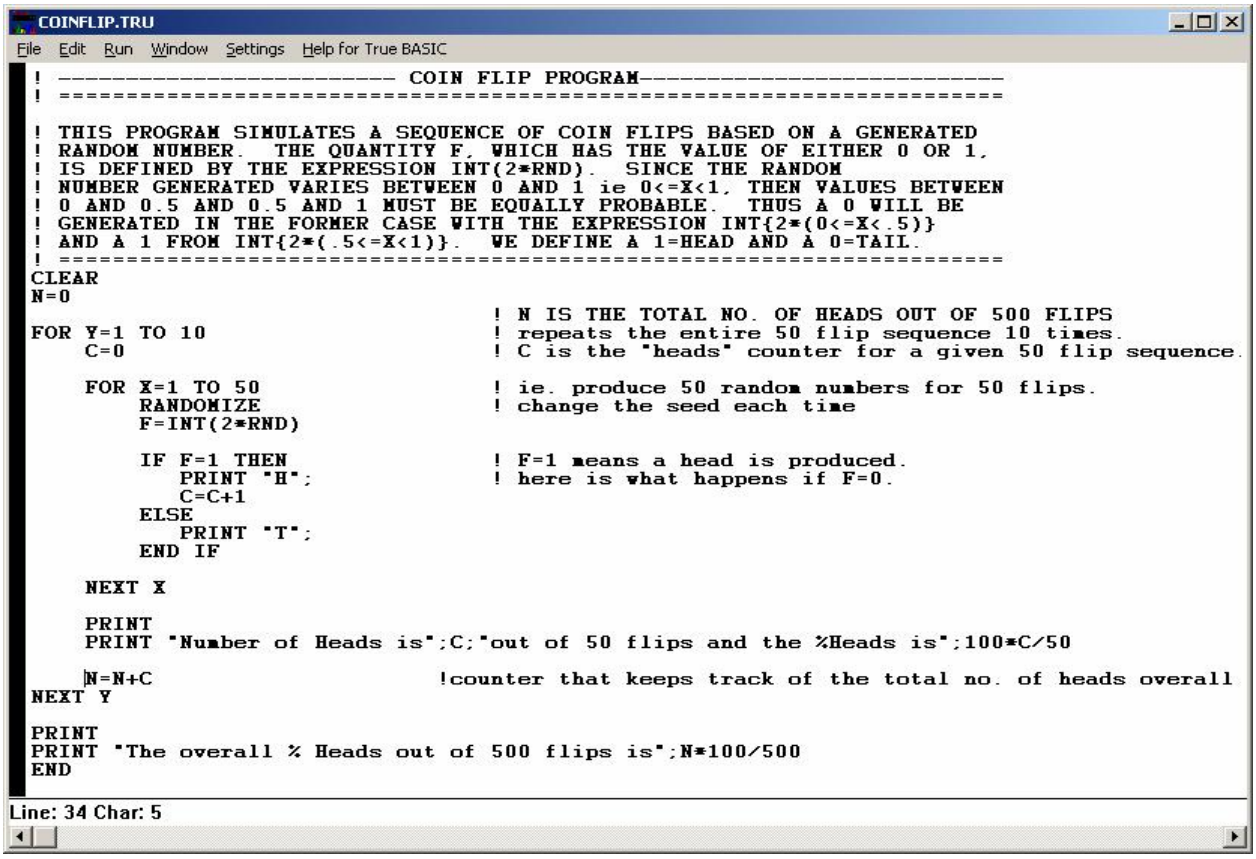
The factorial part is similiar. We start with Factorial initialized to 1 and note that the FOR/NEXT loop is decrementing N, that is, we start with N, then N-1, N-2,... until we reach 1. On the first pass, Factorial=1*I=1*N, on the next pass, Factorial=1*N*(N-1) because I has been decremented by 1, until eventually we have Factorial = 1*N*(N-1)*(N-2)*....*1 which is of course, N!.

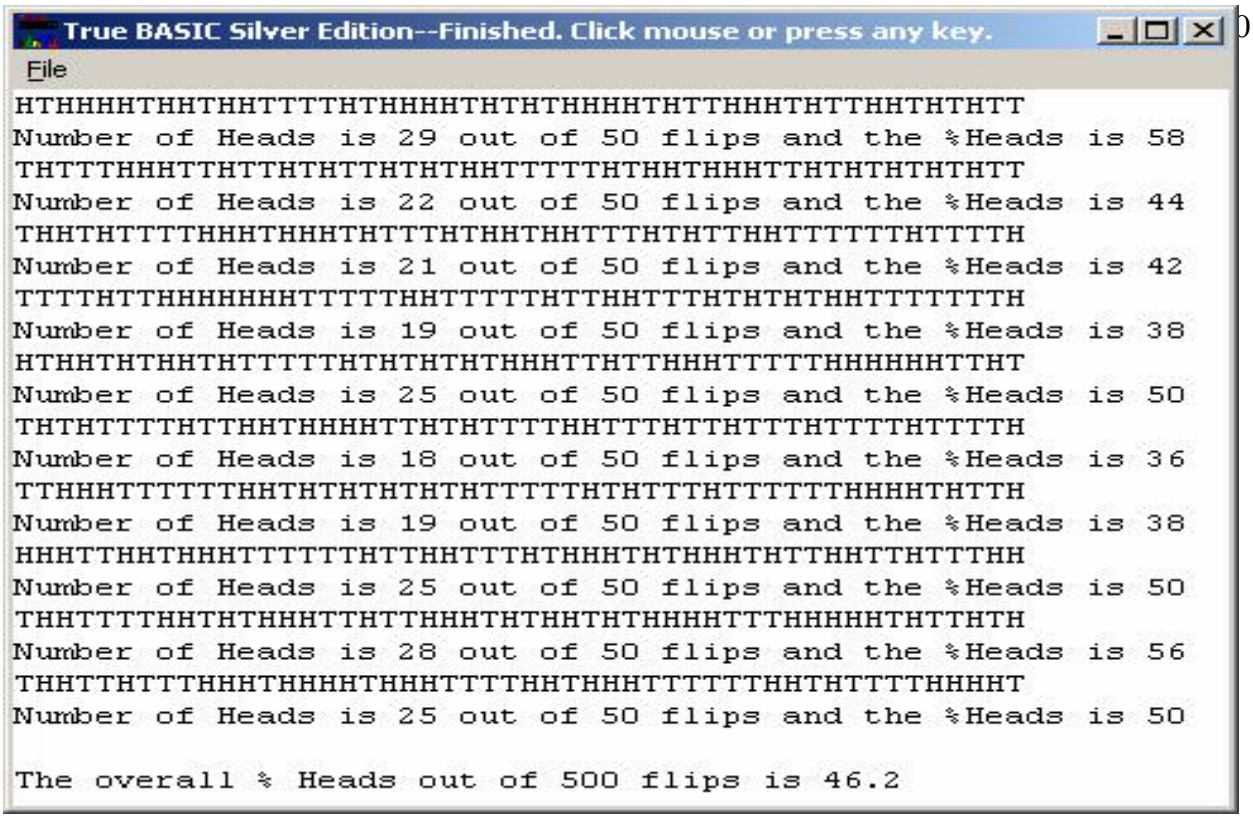
The print statement is straight forward except for the use of the STR$ statement to make N print as a string rather than a number. It serves to remove a space between N and ! which looks better. Delete the STR$ and verify this.

One may also nest **FOR/NEXT** loops as in:

```
FOR i = 1 TO 10
    FOR j = 1 to 5
    ------
    NEXT j
NEXT i
```

This is a common structure, for example, when defining a two dimensional array like a matrix. One must be careful to pair the **FOR** and the **NEXT** statements together as shown. If i and j where interchanged in the above **NEXT** statements a fatal error would occur. Here is a program that uses a nested loop and the random number generator to simulate the flipping of a fair coin along with the output.





The line Coin=INT(2*RND) is the key to how the program works. RND produces a fraction between 0 and 1, that is, $0 \le RND < 1$, so 2*RND will have the range $0 \le 2*RND < 2$. The integer part of this range, i.e., INT(2*RND) will then have only two values, 0 (for $0 \le 2*RND < .5$) and 1 (for $.5 \le 2*RND < 1$) where we define Coin = 1 as a "H" and a 0 as a "T". Since a 0 or 1 will appear with equal probability (for a given flip), then the statement INT(2*RND) is the mathematical equivalent of actually flipping a coin. The second FOR/NEXT loop just repeats the entire flipping process 10 times.

When run, the program produces a string of T's and H's with a print out of the number of heads and their percentage. This program illustrates how the random number generator can be used to simulate a physical process. In fact, computer simulation is often a useful and sometimes essential alternative to experimental measurements in the laboratory or field. We will discuss simulation in more detail later.

PROBLEM: Add the code (3 lines) necessary to keep track of the total number of heads for all flips and then calculate the overall percentage heads.

b**) DO/LOOP**

As we have seen, the DO/LOOP structure acts to repeat the code block between the DO and the LOOP indefinitely. For example, the loop

```
DO
   PRINT "Live long and prosper!";
LOOP
```

will continuously fill the screen with Spock's famous statement until either you stop the program by clicking **STOP** (File Menu), shut the computer off or the computer dies a natual death.

There are however, three non-invasive methods of exiting from the DO/LOOP construct. One of these is the **EXIT DO** statement which can appear anywhere between the DO and the LOOP and redirects execution to the line of code following the LOOP statement. In addition, one can append the conditional statements WHILE and UNTIL to both the DO and the LOOP statements as in,

DO WHILE *condition*
DO UNTIL *condition*
LOOP WHILE *condition*
LOOP UNTIL *condition*

where the *condition* is any logical statement. The distinction between appending WHILE or UNTIL to the DO or to the LOOP is that the DO condition is checked before the loop is executed and if the logical expression is false the loop will not be executed at all, in which case execution drops to the line after the LOOP. The DO/LOOP code block will always be executed once if the *condition* statement is tied to the LOOP.

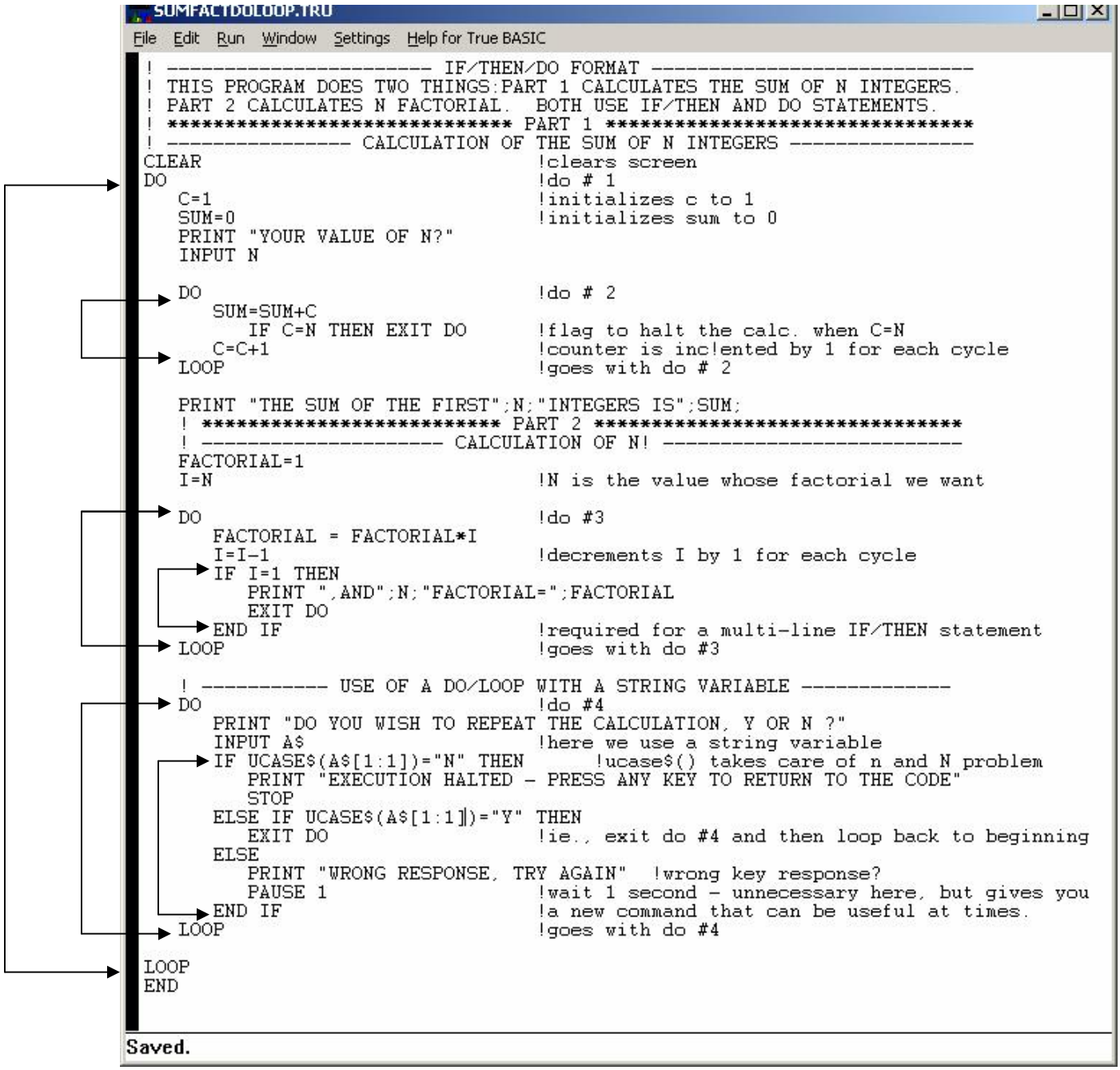
Examples of the *condition* statements to be used with the DO WHILE or LOOP UNTIL statements include:

```
x<=3
S$ <> “NO”
ABS(Y-X) = 0
X<3 AND (Y>0 OR Z<> 2)
(x^2+y^2+z^2) > r^2
(A$ & B$) <> C$
MORE DATA
END DATA
```

that is, any logical statement involving the relational and logical operators is ok. The last two statements are logical clauses and are usually used in conjunction with a READ/DATA statement. These examples should make it clear that the *condition* statement is very flexible which makes the DO/LOOP structure together with WHILE and UNTIL very powerful.

When do we use the DO/LOOP structure? The most obvious case is when we want to repeat a block of code but we don’t know how many times. If we did, we would probably use the FOR/NEXT structure (which is also faster). In addition to repetition of a code block, the DO/LOOP construct is often used, together with an EXIT DO and usually a decision structure like IF/THEN, as a means of providing an early exit from the code block. Here are some examples of the DO/LOOP structure.

Our previous calculation of the sum and factorial of N integers used a **FOR/NEXT** loop because once N was defined; the final loop index was also fixed. Here is the same program using a **DO/LOOP** together with some new statements. Note the importance of indenting the code for readability. This program also illustrates the use of multiple DO/LOOPS for repetition of code blocks as well as the entire program.



One lesson here is that one can frequently use a **DO/LOOP** in place of a **FOR/NEXT** even if they don't know exactly how many repetitions are needed provided there is some other exit criterion that can be attached to the **UNTIL** or **WHILE** statement. However, the FOR/NEXT is often more efficient since there are often fewer statements needed to perform the same computation.

We now discuss the new statements introduced in this program as well as a few others.

UCASE$/LCASE$
These two commands change the case of letters, that is:

PRINT LCASE$("XYZ") produces the string xyz
PRINT UCASE$("xyz") produces the string XYZ

LCASE$ has no affect on lower case characters as does UCASE$ on upper case characters.

You might ask why, in the above program, is it necessary to use the UCASE$ command as part of the Ans$=... statement. It is because “n” and “N” are not the same string (different ASCII codes) and this could cause a problem if the user pressed ‘n’ instead of ‘N’ as asked for. UCASE$ simply forces ‘n’ to be ‘N’. If Ans$ is then a ‘N’, execution passes to the line following the LOOP UNTIL etc. Furthermore, you don’t have to worry about a NO or YES response because A$[1:1] extracts only the first character you type since A$[i:j] functions to identify the substring contained within A$ starting with the i^{th} character and ending with the j^{th}. There is even a better way to do this using the **GET KEY** statement, which as we have already seen, checks the keyboard buffer for each key stroke and captures the corresponding ASCII code for that character. All you then have to do is test to see if the right ASCII code has been entered and go from there.

CHR$/ORD

These functions were listed in the previous function table. CHR$(n), where n is the ASCII value of a keyboard character with the value $0 \leq n \leq 255$, returns the character that n defines. For example, if n = 60, then CHR$(n) = “<“, i.e., the ‘less than’ symbol. Conversly, the ORD(s$) returns the ASCII value of the string character represented by s$ so, for example, ORD(“<“) will return the value 60 and ORD(“m”) = 109.

Here is a short program that uses **GET KEY** along with the CHR$ function to discover what the ASCII code is for any character pressed on the keyboard. This program also illustrates the use of the **EXIT DO** to exit before the loop has finished. To exit at any time press the ESC key whose ASCII value is 27.

CODE FOR ONE CHARACTER AT A TIME

```
PRINT “Press any key sequentially”
   DO
     GET KEY x  !any variable will do, i.e., x, Q etc.
        IF x = 27 THEN EXIT DO   !esc is ASCII 27
     PRINT Chr$(x),x
    LOOP
PRINT  “done”
END
```

ALL 256 CHARACTERS AT ONCE

```
FOR x=0 to 255
   PRINT Chr$(x)
NEXT x
END
```

Alternatively, you could use a **FOR** i = 0 to 255 construct instead of the **DO/LOOP** and **GET KEY** to print the table in one go. Incidentally, the IBM number pad will cause an exception because it uses an ‘extended’ character set. The result is that pressing Home, End etc will cause an exception which crashes the program.

NESTING STRUTURES

FOR/NEXT, DO/LOOP, IF/THEN and SELECT/CASE structures can be nested as needed within a given subroutine without a problem provided the end or closure of each structure is given its proper order of priority in the sense of no overlap as in:

```
DO
  CASE SELECT
    CASE.....
      DO
        FOR .....
          IF ..... THEN
            .....
          ELSE IF   .....
            .....
          END IF
        NEXT
      LOOP
    CASE .....
      .....
   END SELECT
LOOP
```

ARRAYS

We already know that defining a numeric or string variable is tantamount to defining a memory location that symbolizes both the variable name and its current value. True Basic, like most computer languages allows one to define and manipulate a *block* of memory locations where each memory location shares a common name plus a number to identify it. These blocks can be one dimensional like X(1), X(2), ..., X(n) etc. in which case they are called vectors or lists, or two dimensional like X (1,1), X(1,2), ...,X(n,m) where they are called matrices. Here X(1) and X(1,1) etc. are the computer representation of subscripted variables like X_1, and X_{11}, and in fact, the names *subscripted variables* and *arrays* are often used interchangeably. The letter or string part of the name, that is, the X part, can be any valid numeric or string variable name. If the array is a string array, then the '$' must be appended to the array name as in CustName$(1) etc.

The following example illustrates why arrays are useful.

Suppose a teacher wants to average five grades for a class. Without the use of arrays, the teacher might write:

```
PRINT "Input the 5 grades"
INPUT G1,G2,G3,G4,G5
Sum = G1+G2+G3+G4+G5
Average = Sum/5
```

Works OK but it's not very flexible, especially if the class has 250 students in it.

Alternatively, if one knew, for example, that the class would never have more than 20 students, then we could use the simple list array called Grade, and instead write,

```
! Average Grade Program
CLEAR
DIM Grade(20)
   FOR i = 1 to 20
     PRINT "Grade(";i;")= ";
     INPUT Grade(i)
      Sum = Sum+Grade(i)
   NEXT i
 Average = Sum/n
PRINT "Class Average = ";Average
END
```

where now, any number of grades from 1 to 20 can be entered. You should type in the program and run it using some arbitrary percentage scores.

One Dimensional Arrays, Vectors or Lists

The following discussion is limited, for the moment, to one dimensional arrays (or vectors or lists - they are all the same thing) like the one used in the Average Grade program. Understanding that program requires that we first examine the DIM statement.

The Dimension Statement

The DIM or Dimension statement is required whenever one uses an array, and serves to define in computer memory, the name of the array, its dimensions (i.e., one dimensional, two dimensional etc.), its size (subscript range) and the *initial* value of each member of the array. If the array is numeric, each value of the array will be initialized to 0, and if a string variable, to the null string. The DIM statement must appear before any reference to the array otherwise an exception will occur. The syntax is:

```
DIM array name1(array size), array name2(array size), ...
```

For example, one might write:

```
DIM Xcoord(100), Ycoord(100), Z$(5)
```

Here Xcoord and Ycoord are the names of two different subscripted variables, each of which has 100 memory locations allocated to it, Z$ is a string variable with 5 locations allocated. Thus you pick the array name but the DIM statement assigns the subscripts (sequentially) and the initial array values. In addition to assignments like DIM Y(5) where the default least subscript is 1, True Basic also permits you to define the lower bound for the subscript with the notation,

DIM X(lower bound:upperbound)

where the lower and upper bounds can be any signed (that is, negative or positive) integer.

For example, DIM X(-5:10) where now the least subscript is X_{-5} and the largest is X_{10}. Attempts to access any X with a subscript less than -5 or greater than 10 will result in a 'subscript out of bounds' error. The ':' in the above DIM statement can be replaced by the word 'TO'. As a less attractive alternative to defining a lower bound with the preceeding syntax, one can use the **OPTION BASE** statement. For example,

```
OPTION BASE 0
DIM X(10)
```

will define the range of the list as X_0, X_1, ... , X_9. The equivalent DIM statement would be DIM X(0:9) or DIM X(0 to 9). Like the bounds, the argument of the **OPTION BASE** statement can be any signed integer. That said, we do not recommend changing the default option base, but rather use the DIM x[i:j] syntax.

Redimensioning a Matrix

In the Average Grade program, DIM Grade(20) tells the computer to set aside in memory enough room for a one dimensional list consisting of 20 variables, Grade(1),...Grade(20). In general however, using a fixed size allocation is often wasteful of memory (each number allocated takes 8 bytes or 64 bits in memory) if you don't always need the whole allocation, and in any case you lose flexibility when you lock the program into an array of fixed size. It often makes more sense to define the array for a single entry as in DIM Grade(1) and then redefine the size of the list once you know what your needs will be while the program executes. The procedure for changing the array size 'on the fly' is to use the *MAT REDIM* statement which allows one to change the size of the array during program execution (but not the dimensions - that is, you cannot change a one dimensional array into a two dimensional array during execution). The syntax is:

MAT REDIM *array name*(new array size)

For example, in the Average Grade Program, we fixed the array size at 20. Instead, suppose we replace the DIM Grade(20) with the three statements:

```
DIM Grade(1)
INPUT PROMPT "How many grades ?":n
MAT REDIM Grade(n)
```

where now we begin by defining the list to be of size 1, that is, a single memory location. We then ask the user how many grades they want to input and store this result as n. The MAT REDIM Grade(n) now increases the number of memory locations by a factor of n. For example, if there are 250 students then n = 250 and MAT REDIM Grade(n) defines an additional 249 locations, Grade(1), Grade(2), ..., Grade(250). Each of these 250 different numeric variables will then hold a single student grade when inputed from the keyboard. You should add the above statements to the Average Grade program and confirm that you can now add any number of grades up to n.

Finally, the indicies must be integers, in other words, you cannot define a subscripted variable to be X(0.2) or X(1.9) any more than you would write $X_{0.2}$ or $X_{1.9}$ mathematically. Non-integer subscripts will be rounded.

Here are some useful list operations.

BASIC LIST OPERATIONS

Defining a List:

1) READ/DATA STATEMENT

```
DIM P(4)
FOR i=1 TO 4
   READ P(i)
NEXT i
DATA 1,2,3,4
```

Alternatively, *MAT READ P* as in:

```
 DIM P(4)
 MAT READ P
 DATA 1,2,3,4
```

accomplishes the same task even more simply.

2) INPUT STATEMENT

```
DIM P(4)
FOR i=1 TO 4
   INPUT P(i)   !each pass puts a new ? on the screen
NEXT i
```

Again, a more efficient way to input data is with the *MAT INPUT* statement, e.g.,

```
DIM P(4)
MAT INPUT P
```

will cause the program to pause for the user to enter 4 items, separated by commas, from the keyboard. Note that this method does require that the user be prompted for the value of n.

3) ARITHMETIC EXPRESSION

```
DIM P(100)
FOR k=1 TO 100
    P(k)=100*k^2   !note that the expression must involve k
NEXT k
```

Examples 1 and 2 work for strings as well - note that the items in the DATA statement do not have to be enclosed in quotes, as in,

4) DIM DayOfWeek$(7)

```
FOR j=1 TO 7
       READ DayOfWeek$(j)
NEXT j
DATA Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday
```

The DO/LOOP can also be used when the number of input items is not known, e.g.,

```
DIM P(1)
 i=1
    DO WHILE MORE DATA
    READ P(i)
     i=i+1
    MAT REDIM P(i)
   LOOP
DATA 1,2,3,4
MAT PRINT P  !prints the array
END
```

but if you run this program you will get 1,2,3,4,0 on screen where the 0 comes from the fact that i = 5 when the loop is executed for the final time (4th data value) so that when MAT REDIM P(5) is executed we will

get its initialized value of 0. These potential problems illustrate that you must always verify your output values and never simply assume that what looks ok *is ok*. In fact, loops like this can get tricky because the loop index is often one more or one less than programmer's impeccable logic would suggest. The key to the problem is to note that the index i in Read P(i) is always one less than the i in the Mat Redim P(i) statement.

The solution to the problem is to make sure that the READ and the MAT REDIM statements march in lock step as to the value of the index i. To do this, rewrite the above code to read,

```
DIM P(1)
 i=0
      DO WHILE MORE DATA
       i=i+1
      MAT REDIM P(i)
      READ P(i)
     LOOP
DATA 1,2,3,4
MAT PRINT P
END
```

This time the offending 0 will be gone.

Now try moving the READ P(i) to the line above the MAT REDIM P(i) statement. You will get a fatal error ('subscript out of bounds'). Why ? The reason is that you must redimension P(i) *before* you assign it a value in the READ statement, not *afterwards* -- i.e., you can't store an array or list element in memory before you make room for it.

What happens in the following two code fragments, i.e., do they work and if not, why not ?

```
DO                  FOR I=1 TO 4
   READ X(I)           READ X
LOOP                NEXT I
DATA 1,2,3,4        DATA 1,2,3,4
```

Printing a List:
A list may be printed with a simple FOR/NEXT loop as in,

```
FOR I=1 TO 4
   PRINT X(I)
NEXT I
```

or alternatively, using the *MAT PRINT X* statement (note - not X(I)). There is also a MAT PRINT USING statement, as in.

```
DIM x(5)
MAT INPUT x
MAT PRINT x
END
```

MAT statements are extremely compact and strongly recommended when possible.

Relocating Items in a List:
Suppose we have a list where P(1)=4 and P(2)=2 and want to exchange these values. The code is simple,

```
 z = P(1)
 P(1) = P(2)
 P(2) = z
```

This swap routine forms the basis of the 'bubble sort' which we will discuss shortly.

Accumulating a Sum:

Summing a list is just a variant on the usual statement *sum = sum+x*. For example, suppose we wish to sum the first 10 integers. The following code does just that.

```
DIM D(10)
 SUM = 0
   FOR i=1 TO 10
      READ D(i)
      SUM = SUM + D(i)
   NEXT i
DATA 1,2,3,4,5,6,7,8,9,10
```

Note that since we are dealing with consecutive integers, the READ/DATA statements could be replaced by the statement D(i) = i.

Counting Members of a List:

There are times, like reading elements of a file or a long DATA statement, that one needs to know how many members there are in the list. Here is the procedure.

```
DO WHILE MORE DATA
      READ x   !x is not a subscripted variable so its value changes after each loop
      Count = Count +1
LOOP
DATA etc.
```

Count is then the number of items in the file or DATA statement. This number could then be used later in the program, as in, for example, a FOR/NEXT loop.

Finding the Maximum (or Minimum) Member of a List:

The following code fragment will find the least and greatest number in a list of numbers.

```
  S = X(1)
  FOR i = 2 TO N   !N assumed known - see previous algorithm
    IF X(i) > S THEN  S = X(i)
  NEXT i
```

The final value of S will be the maximum value in the list. How do we get the least value ?

Count Frequency of Integer Elements in a List

Suppose we have a DATA statement consisting of a set of repeating integers, e.g.,
DATA 0,2,1,3,0,2,4,.. The following algorithm counts the number of times, or frequency, that each integer appears.

```
DO WHILE MORE DATA
      READ i    !note that it is the integer index that is being read and its the frequency that is counted
      Freq(i) = Freq(i) + 1
LOOP
DATA etc.
```

Before this can be run, Frequency must be dimensioned to the maximum of unique integers in the list, that is, suppose the DATA statement read, 0,2,6,3,6,6,3. Then Frequency would have to be dimensioned to 4 corresponding to 0,2,3 and 6. Furthermore, it would be necessary to use OPTION BASE 0 to accommodate the 0 value. In this example, we would find that Freq(0)=1, Freq(2)=1, Freq(3)=2, and Freq(6)=3. Here is a program that illustrates how this code frag works. The code also illustrates why one should use meaningful variable names instead of single letter names like T to represent the number of families with i TV sets.

TABULATE.TRU

File Edit Run Window Settings Help for True BASIC

```
!------------------------ TABULATION ROUTINE -------------------------------
!THIS PROGRAM TABULATES THE NUMBER AND PERCENTAGES OF FAMILIES WHO OWN
!0,1,2,3 AND 4 TV SETS.  THIRTY THREE FAMILIES WERE POLLED.
OPTION BASE 0                   !remember, the default base is 1
DIM T(5)                        !T(I) is the no. of families with I tv sets
DIM P(5)                        !P(I) is the percentage of families with I tv sets
DIM N(35)                       !N(I) is the subscripted variable that defines the data

LET C=0                         !C counts the numbers of families in the read/data loop
DO WHILE MORE DATA
   READ K
   C=C+1                        !C counts the number of families and equals 33.
   T(K)=T(K)+1                  !counter for the number of families with N sets
LOOP

PRINT TAB(1);"THE NO. OF TV!S";TAB(28);"NO. OF FAMILIES";TAB(56);"PERCENTAGE"
PRINT TAB(1);"---------------";TAB(28);"---------------";TAB(56);"----------"

FOR I=0 TO 4
    P(I)=T(I)*100/C             !percentage
    PRINT TAB (7);I;TAB (34);T(I);TAB (59);
    PRINT USING "##.##":P(I)
NEXT I

!Here we calculate the maximum number of tv's owned by any one family - this
!is the algorithm for picking the max (min) term out of a list of numbers
S=0                             !S is a comparator, its initial value is the 0
RESTORE
FOR I=1 TO C
    READ N(I)                   !i.e. N(1)=1,N(2)=0,...,N(33)=1 etc
    IF N(I)>S THEN S=N(I)       !sets S=N when N exceeds the current S
NEXT I
Print
PRINT "THE MOST TV!S OWNED BY ANY ONE FAMILY IS";S;

!we now ask, how many familes owned this maximum number of tv sets?

J=0                             !counter
FOR I=1 TO C
    IF N(I)=S THEN J=J+1        !only count N(I) when it is maximum
NEXT I
PRINT "AND";J;"FAMILIES HAVE THIS MANY."

DATA 1,0,3,2,1,2,0,1,4,1,1,2,4,1,3,1,2,2,1,3,1,2,2,1,1,2,3,4,3,1,1,1,1
END
```

Line: 45 Char: 5

True BASIC Bronze Edition--Finished. Click mouse or press any key.

File

```
THE NO. OF TV!S            NO. OF FAMILIES            PERCENTAGE
---------------            ---------------            ----------
       0                          2                       6.06
       1                          15                     45.45
       2                          8                      24.24
       3                          5                      15.15
       4                          3                       9.09

THE MOST TV!S OWNED BY ANY ONE FAMILY IS 4 AND 3 FAMILIES HAVE THIS MANY.
```

Creating a New List f rom an Old One.

Here we generate a new list consisting of selected elements from a previous list. Assume the new list is to contain only those values of the old list greater than 10.

```
 k = 1
  FOR i = 1 TO N
    IF A(i) > 10 THEN
       B(k) = A(i)    !A is the old list, B the new one composed of values > 10
       k = k + 1      !B index
    END IF
NEXT i
```

PRIME NUMBER PROGRAM

Here is a program that computes prime numbers and stores them as an array. As prime number algorithms go, this program is pretty slow and inefficient, but it makes up for that with understandability and for our purposes, that's a fair trade off.

PRIME Simple.TRU

File Edit Run Window Settings Help for True BASIC

```
!-------------------------------- Simple Prime --------------------------------
!PROGRAM TO FIND PRIME NUMBERS <= A GIVEN N. COMPARE THIS WITH THE PROGRAM
!PRIME#1 WITH USES A SIEVE AND IS WHICH IS ABOUT 4 TIMES FASTER THAN THIS
!ALGORITHM.

DIM PRIME(1000)

PRINT "WHAT IS N?"
INPUT N

LET TI=TIME                          !time the algorithm

LET K=1

FOR J=2 TO N                         !J generates the set of integers from 2 to N
    FOR I=2 TO INT(SQR(J))           !all prime factors fall between 2 and sqr(j)
        IF MOD(J/I,1)=0 THEN         !found a factor, ie, no remainder so not prime
           EXIT FOR                  !exits first FOR, then executes NEXT J
        ELSE IF I=INT(SQR(J)) THEN   !no factors found
           PRIME(K)=J                !store the Kth prime
           K=K+1                     !inc. K for the next prime
           EXIT FOR
        END IF
    NEXT I
NEXT J

LET TF=TIME
LET TIME_INTERVAL=TF-TI

PRINT "2  3 ";                       !routine does not handle 2 or 3

FOR I=1 TO K-1
    PRINT PRIME(I);
NEXT I

PRINT
PRINT "TIME=";TIME_INTERVAL

END
```

Note that the EXIT FOR moves execution to the NEXT J and skips the NEXT I

Saved.

True BASIC Bronze Edition--Finished. Click mouse or press any key.

File

```
WHAT IS N?
? 1000
2  3  5  7  11  13  17  19  23  29  31  37  41  43  47  53  59  61  67  71  73  79  83  89  97  101  103  107  109
 113  127  131  137  139  149  151  157  163  167  173  179  181  191  193  197  199  211  223  227  229  233  239
 241  251  257  263  269  271  277  281  283  293  307  311  313  317  331  337  347  349  353  359  367  373  379
 383  389  397  401  409  419  421  431  433  439  443  449  457  461  463  467  479  487  491  499  503  509  521
 523  541  547  557  563  569  571  577  587  593  599  601  607  613  617  619  631  641  643  647  653  659  661
 673  677  683  691  701  709  719  727  733  739  743  751  757  761  769  773  787  797  809  811  821  823  827
 829  839  853  857  859  863  877  881  883  887  907  911  919  929  937  941  947  953  967  971  977  983  991
 997
TIME=  .01
```

The algorithm is straight forward. We test each value of j from 3 (2 is prime) to 1000 for a factor by dividing j by i from 2 to the square root of j. The test is the MOD statement which is 0 if a factor is found. If no factor is found for any value of i then j is prime and we make a list, called Prime, of those values. Notice also that the **EXIT FOR** takes you out of the first loop, not the second.

BUBBLE SORT PROGRAM

This program uses the swap routine to sort a set of numbers in ascending rank. This sort method is called a 'Bubble Sort' and is probably the least efficient sorting method available, but for small set of numbers it works fine and is fairly easy to understand.

BUBBLESORT.TRU

File Edit Run Window Settings Help for True BASIC

```
!------------------------- BUBBLE SORT ---------------------------
!THE "BUBBLE SORT" WILL ORDER A SET OF NUMBERS IN ASCENDING RANK.
DIM L(5)  !option base 1 is default

MAT READ L

 PRINT "Here is the original list"
 MAT PRINT L;
PRINT

PRINT  "We now arrange the list stepwise"
PRINT

FOR I=1 TO 4            !I and J are compared until L(I)>L(J) then swapped
   FOR J=I+1 TO 5       !Note that we never compare the same elements
      IF L(I)>L(J) THEN
         S=L(I)         !|1
         L(I)=L(J)      !|2    swap routine
         L(J)=S         !|3
         PRINT "I=";I;"L(I)=";L(I);"J=";J;"L(J)=";L(J) !I changes when its current value is least
         MAT PRINT L;
      END IF
   NEXT J
NEXT I

 PRINT
PRINT "Here is the ordered list"
MAT PRINT L;

 DATA 6,-50,-200,.001,5
```

Running...

True BASIC Bronze Edition--Finished. Click mouse or press any key.

File

```
Here is the original list
 6 -50 -200  .001  5


We now arrange the list stepwise

I= 1 L(I)=-50 J= 2 L(J)= 6
-50  6 -200  .001  5

I= 1 L(I)=-200 J= 3 L(J)=-50
-200  6 -50  .001  5

I= 2 L(I)=-50 J= 3 L(J)= 6
-200 -50  6  .001  5

I= 3 L(I)= .001 J= 4 L(J)= 6
-200 -50  .001  6  5

I= 4 L(I)= 5 J= 5 L(J)= 6
-200 -50  .001  5  6


Here is the ordered list
-200 -50  .001  5  6
```

The program starts by comparing the first element (I=1) with the second element (J=2) where the **IF** statement is found 'true' because 6 is greater than -50 so we swap elements, i.e., L(1) becomes -50 and L(2) becomes 6. L(1) is again compared with L(3) and again, elements are swapped so now L(1) is -200 and L(3) is -50. Only after the least value is found is I incremented to 2 and the comparisons begin all over. Obvious inefficiencies abound due to a large number of redundant comparisons. In fact, the Bubble Sort is an N^2 routine, i.e., time to sort is proportional to the number of items (N^2). Whether or not a bubble sort suits your purpose obviously depends on the speed of your computer and the number of elements to be sorted, but in general sorting routines like the Heapsort, Quicksort, Mergesort, Shellsort etc., would be preferable for anything other than small N (100 or less).

Two Dimensional Arrays (Matrices or Tables)

A two dimensinal array, or matrix, is used when we have data that depends on two attributes. Suppose, for example, you measured the rate of a reaction as a function of both pH (integer values between 8 and 10, say) and the temperature at 1 degree intervals from 20^0C to 25^0C. Then to each value of the rate you would have a corresponding temperature and a pH, and the natural way to display these results would be to use a table like,

	pH →		
T	Rate(20,8)	Rate(20,9)	Rate(20,10)
↓	Rate(21,8)	Rate(21,9)	.
	Rate(22,8)	.	.
	.	.	.
	.	.	.
	Rate(25,8)	.	Rate(25,10)

where each element of the table is a rate which is a function of both pH (across) and the temperature (down), and where the number in parentheses is an index that corresponds to the value of the pH and temperature for that rate. In other words, we have a matrix where the rates are the matrix elements and the numbers in parentheses are the indicies that identify each element of the matrix according to the general format,

ArrayName(row subscript,col. subscript)

Thus the element A(23,10) would be the rate when T = 23^0C and the pH = 10. The fact that the indices in the 'rate table' correspond to physical values of the variables pH and Temperature is convenient, but unnecessary. The subscripts could have just as well been the usual integers 1,1; 1,2; ... ; 5,3.

The Dimension Statement

Like the one dimensional array, the two dimensional array is defined once the DIM statement is executed. The syntax is,

DIM *ArrayName*(d_1,d_2)

where d_1 is the number of row elements and d_2 is the number of column elements.

For example DIM X(4,3) defines a 4-by-3 (i.e., 4 rows and 3 columns) matrix consisting of 12 elements total, and where each value is initialized to 0.

One can also define the lower and upper indicies for the subscripts using the snytax,

DIM *ArrayName*(Lrow index to Urow index,Lcol index to Ucol index)

where Lrow and Urow etc. refer to lower row and upper row.

Thus the dimension statement for the 'rate matrix' would be

DIM Rate(20 to 25,8 to 10)

if one chose to have the indices represent the pH and Temperature directly.

Redimensioning of two dimensional arrays can be accomplished with the statement,

MAT REDIM A(d_1,d_2)

Changing the size of a two dimensional array 'on the fly' can be tricky and data will be lost if the new matrix is of lower dimensionality. Be careful to check the resulting matrix.

INPUT/OUTPUT COMMANDS FOR TWO DIMENSIONAL ARRAYS

Exactly the same commands are used for two dimensional arrays as were used for one dimensional lists.

To construct the 'rate matrix' of 18 values we would use,

```
DIM Rate(25:30,8:10)
  FOR Row = 25 TO 30
    FOR Col = 8 TO 10
      INPUT Rate(Row,Col)
    NEXT Col
  NEXT Row
END
```

or, alternatively,

```
DIM Rate(25:30,8:10)
  MAT INPUT Rate
END
```

In the former case you will get 18 question marks, one after the other as you input each value of the rate, whereas in the latter case, you get a single question mark which expects you to input 18 values, each separated by a comma. Apart from the details of the input, the resulting matrices will be the same.

It is important to understand how a double FOR/NEXT statement works. In the case above, the outside FOR/NEXT begins by setting Row = 25 after which the inside FOR/NEXT is executed completely, i.e., for Col = 8, 9 and 10. Flow then passes back to the outside FOR/NEXT whereupon Row is set equal to 26 and the inside FOR/NEXT is again executed for all three values of Col. In this way, the rows are written sequentially.

Here is a simple program that constructs a 5 by 6 matrix and then computes the row sums and the sums of those sums.

MATRIX.TRU

File Edit Run Window Settings Help for True BASIC

```
! ---------------------------- ARRAYS -----------------------------
! THIS PGM. GENERATES A TABLE OR MATRIX AND THEN COMPUTES ROW SUMS
DIM Mat(5,6)

NumberRows=5
NumberCols=6

FOR i=1 to NumberRows
    FOR j=1 to NumberCols
        READ Mat(i,j)              !here'RowSum the Mat
    NEXT j
NEXT i

!which we print
FOR i=1 to NumberRows
    FOR j=1 to NumberCols
        PRINT Tab(4*j);Mat(i,j);   !horizontal spacing
    NEXT j

    PRINT                          !vertical spacing
    PRINT
NEXT i

!compute the row sums
DIM RowSum(5)

FOR i=1 to NumberRows
    FOR j=1 to NumberCols
        RowSum(i)=RowSum(i) + Mat(i,j)
    NEXT j

    PRINT "Row(";i;") ";"RowSum=";RowSum(i)
NEXT i

!compute the RowSum of the row sums
SumOfSums =0

FOR i=1 to 5
    SumOfSums=SumOfSums+RowSum(i)
NEXT i

PRINT
PRINT "The sum of the RowSums is"; SumOfSums

DATA 5,6,6,0,-12,4,17,21,-8,15,5,5,-18,0,11,3,1,-17,12,7,13,2,13,-9
DATA 24,4,-27,-3,0,14,8,-10
END
```

Run successfully.

True BASIC Bronze Editio...

File

```
     5    6    6    0   -12   4

     17   21  -8    15   5    5

    -18   0    11   3    1   -17

     12   7    13   2    13  -9

     24   4   -27  -3    0    14

Row( 1 ) RowSum= 9
Row( 2 ) RowSum= 55
Row( 3 ) RowSum=-20
Row( 4 ) RowSum= 38
Row( 5 ) RowSum= 12

The sum of the RowSums is 94
```

MATRIX COMMANDS

True Basic supports a rich array of matrix commands thus obviating the need for extensive use of nested FOR/NEXT loops. By way of a simple example, suppose we wish to multiply the following two matrices:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} 6 & 8 \\ 9 & 0 \\ 1 & 2 \end{pmatrix}$$

in which case we find,

$$C = A * B = \begin{pmatrix} 27 & 14 \\ 75 & 44 \end{pmatrix}$$

where the code to multiply these two matrices is based on the algorithm for computing the c_{ij} element of the product matrix C,

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Here, if A is an m-by-n matrix and B is a n-by-l matrix then the product matrix, C, is an m-by-l matrix.

The code that implements this multiplication, assuming the A and B matrices have already been defined (e.g., with a MAT READ or MAT INPUT statement together with appropriate DIM statements for A, B and C) is the following.

```
FOR i = 1 to 2
  FOR j = 1 to 2
    Sum = 0
      FOR k = 1 to 3
          Sum = Sum + A(i,k)* B(k,j)
       NEXT k
        c(i,j) = Sum
  NEXT j
NEXT i
```

With True Basic's MAT commands, these 9 lines of code can be replaced by the single statement MAT C = A*B. However, there is always a tradeoff - the MAT command takes about 1.7 times longer to execute than the nested FOR/NEXT loops.

The following table lists the most common matrix commands supported by the language.

TRUE BASIC MATRIX COMMANDS

OPERATION	COMMENTS
MAT A = B ± C	A and B must be the same size
MAT B = k * A	Multiplication by a scalar k
MAT C = A * B	A must have the same number of columns as B has rows
MAT A = ZER(n,m)	Redimensioning of A to a n-by-m matrix with all elements set to zero
MAT A = k*CON(n,m)	Defines a n-by-m matrix with each element equal to k. CON by itself defines each element as 1.
MAT A = IDN(n,n)	n-by-n identity matrix (matrix must be square)
MAT A = TRN(B)	Transpose of B
MAT A = INV(B)	Inverse of B (B must be square)
MAT A$ = NUL$(n,m)	String equivalent of MAT A = ZER(n,m). Each element is the null string "".
DOT (A,B)	Scalar (dot) product of vectors A and B
DET A	Determinant of A (A must be square)

In addition to these commands, there are the following statements used to determine the size and index bounds of a matrix.

SUBSCRIPT RANGE COMMANDS

OPERATION	COMMENTS
LBOUND(A,d)	Returns least subscript in the dimension d
UBOUND(A,d)	Returns maximum subscript in the dimension d
SIZE(A,d)	Returns the total number of elements in the array A in dimension d. If d is omitted for a d_1-by-d_2-by- ... d_{n-1}-by-d_n dimensional array, then SIZE(A) returns the product $d_1 * d_2 * ... * d_n$.

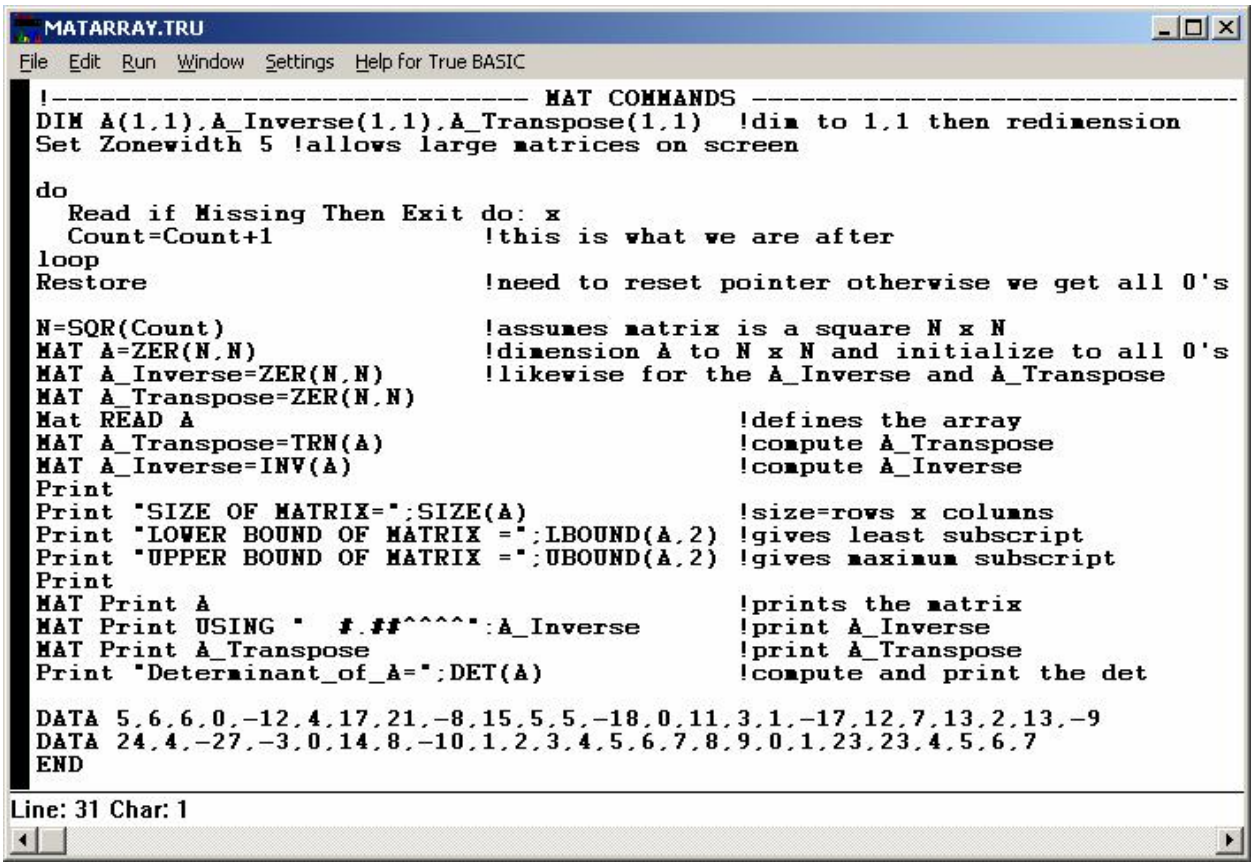
The value these functions return are determined from the dimension statement for the array A. The dimension, d, can be omitted for a one dimensional list, but must be included for arrays of two or more dimensions. For arrays of two or more dimensions, d can vary from 1 to the full dimensionality of the array. For example, in the three dimensional array A(3,4,5), d=1 refers to the row subscripts from 1 to 3, d=2 refers to the column subscripts 1 to 4, and d=3 refers the subscripts from 1 to 5.
A few examples will help clearify how the range functions work.

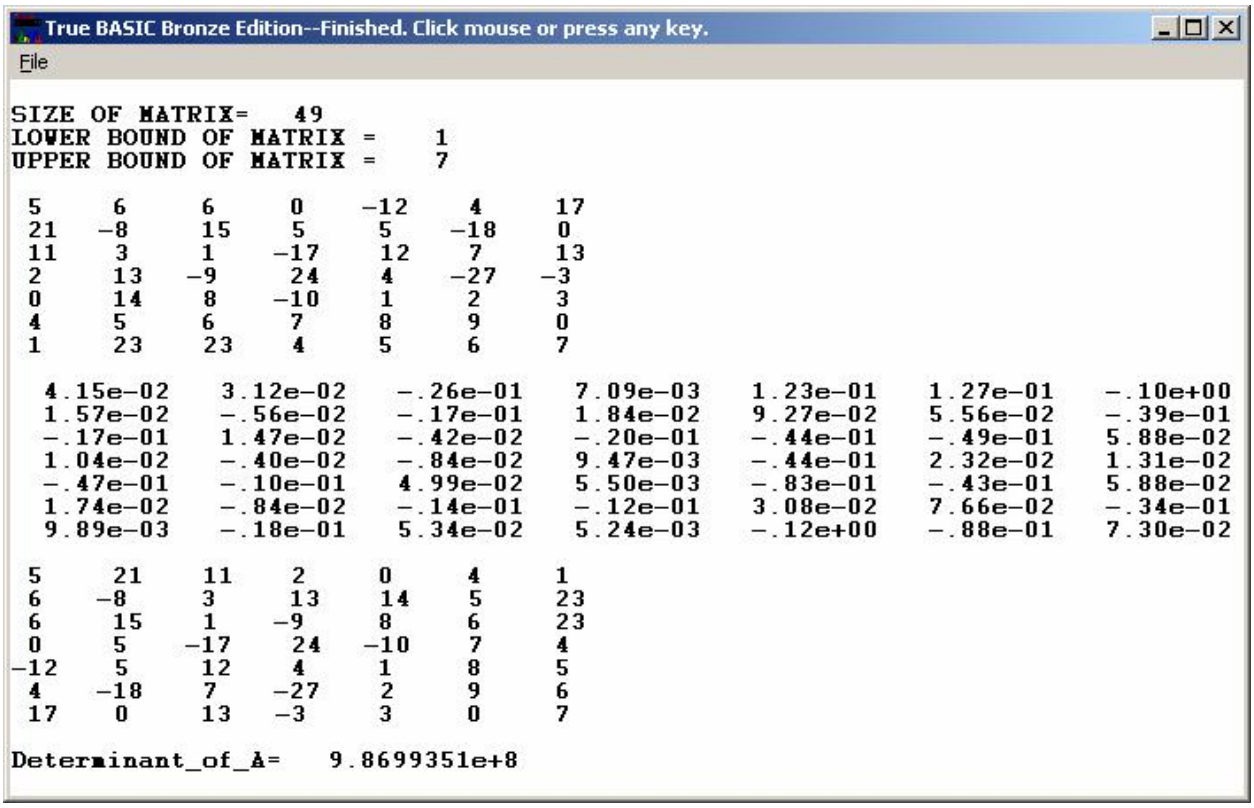
EXAMPLES OF SUBSCRIPT RANGE FUNCTIONS

DIM STATEMENT	Size	d	Size(d)	Lbound(A,d)	Ubound(A,d)
DIM A(6)	6	1	6	1	6
DIM (0 to 10)	11	1	11	0	10
DIM(2,7)	14	2	7	1	7
DIM(3 to 10,5)	40	1	8	3	10
DIM A(3,4,6,5 to 9)	360	4	5	5	9

OTHER MAT COMMANDS

We have already encountered the **MAT INPUT**, **MAT READ** and **MAT PRINT** statements. Both the **MAT INPUT** and **MAT PRINT** have alternatives that give the users more control. These include the **MAT PRINT USING**, **MAT LINE INPUT** and **MAT INPUT PROMPT**. Read your text for details. The following program and output illustrates the use of some of these MAT operations.




True BASIC Bronze Edition--Finished. Click mouse or press any key.
File
SIZE OF MATRIX= 49
LOWER BOUND OF MATRIX = 1
UPPER BOUND OF MATRIX = 7
5 6 6 0 -12 4 17
21 -8 15 5 5 -18 0
11 3 1 -17 12 7 13
2 13 -9 24 4 -27 -3
0 14 8 -10 1 2 3
4 5 6 7 8 9 0
1 23 23 4 5 6 7
4.15e-02 3.12e-02 -.26e-01 7.09e-03 1.23e-01 1.27e-01 -.10e+00
1.57e-02 -.56e-02 -.17e-01 1.84e-02 9.27e-02 5.56e-02 -.39e-01
-.17e-01 1.47e-02 -.42e-02 -.20e-01 -.44e-01 -.49e-01 5.88e-02
1.04e-02 -.40e-02 -.84e-02 9.47e-03 -.44e-01 2.32e-02 1.31e-02
-.47e-01 -.10e-01 4.99e-02 5.50e-03 -.83e-01 -.43e-01 5.88e-02
1.74e-02 -.84e-02 -.14e-01 -.12e-01 3.08e-02 7.66e-02 -.34e-01
9.89e-03 -.18e-01 5.34e-02 5.24e-03 -.12e+00 -.88e-01 7.30e-02
5 21 11 2 0 4 1
6 -8 3 13 14 5 23
6 15 1 -9 8 6 23
0 5 -17 24 -10 7 4
-12 5 12 4 1 8 5
4 -18 7 -27 2 9 6
17 0 13 -3 3 0 7
Determinant_of_A= 9.8699351e+8

PROCEDURES

So far our programs have been written without much apparent structure other than the required order in which statements had to appear to avoid runtime errors. This is okay for a program of a few dozen lines of code, but when the length runs to multiple pages and there are numerous decision branches, then it makes sense to look for ways to package parts of the program into units which accomplish specific tasks and which can be invoked by the main program when needed. These sub programs units are termed *procedures* and include both *subroutines* and *functions*.

Apart from the logistical and esthetic advantages of reducing code to managable units, there is also the issue of code repetition. For example, suppose one has written a program that requires, amongst other things, that the factorial of a number be computed numerous times before the program terminates. Without subroutines, the code to compute the factorial would have to be repeated in full each time it is needed. Instead, if the factorial code was contained in a single procedure, either a subroutine or a function, and which could be implemented when desired, then the resulting code would be much smaller and cleaner.

In this example, the factorial procedure does only one thing - compute the factorial of a number. A single subroutine or function can be designed to do as many tasks as desired, but multiple tasks defeat the purpose and advantage of using them. Ideally, 'one task - one procedure' makes good programming sense.

SUBROUTINES

Types of Subroutines

There are two types of subroutines, *internal* and *external* and they are easily distinguished between. If the last statement in a program containing subroutines is the END statement, then all subroutines are, by definition, *internal*, and are referred to as *internal* subroutines (or subs).

Subroutines that are external to the **END** statement (that is, come after the **END** statement), are called *external*.

For example,

```
!These are internal subroutines
some code
Subroutine 1
Subroutine 2
....
Subroutine n
END
```

```
!These are external subroutines
some code
END
 Subroutine 1
Subroutine 2
      ....
Subroutine n
```

Which is which should be obvious - look for the END statement.

Subroutine Syntax

There are two parts associated with implementing any subroutine. These include,

a) the CALL statement
   Syntax: CALL *subroutine name* (parameter list)
b) subroutine definition
   Syntax:
   SUB *subroutine name* (parameter list)
   code that defines the subroutine task
   END SUB

The *subroutine name* must be identical in the CALL and SUB statements, and, like the naming of variables, the subroutine name ought to reflect what the subroutine does.

Execution Flow and the EXIT SUB Statement

When a sub is called, program flow branches to the called subroutine which is then executed until either the program flow terminates naturally with the END SUB statement or, before reaching the END SUB with an ***EXIT SUB*** statment. Flow then returns to the next line of code after the previous CALL statement and execution continues.

Local and Global Variables and Their Scope

In the context of subroutines, string or numeric variables (or data), are said to be either *local* or *global*. A *global* variable is one that is known or is accessible from anywhere within a program whereas a *local* variable is one whose existence is known only within the subroutine in which it is used or defined. A way to qualitatively refer to this 'visiability' of variables is to use the word *scope*: a *global* variable is one with a broad *scope* while a *local* variable has a restricted *scope*.

The Parameter List

The parameters in the parameter list are the numeric, string, or array variables needed for one subroutine to communicate with one another. A parameter list is optional for ***internal*** subroutines, but if it is omitted, then all variables and data defined or used in the subroutine will be global (i.e., known) to the rest of the program including all of the other subroutines. Internal subs may or may not have a parameter list. Internal subs without a parameter list are an invitation to disaster since all variables will be global and hence subject to unintended corruption. Internal subroutines with parameter lists are safe enough in principle, but if you are going to write subroutines with parameter lists you might as well use external subs where all variables are local to the subroutine and less likely to be corrupted. Furthermore, if one moves onto other languages like Visual Basic, internal subroutines won't even be an option so you might as well do things properly to start with.

Programming Tip

The whole idea here is to limit the exposure or visibility that the program has to any variable; in other words, always try to minimize variable scope (i.e., visibility), and that is most readily done by using external subroutines.

External Subroutines

A parameter list is ***required*** when working with an ***external*** subroutine since all variables are strictly local to that subroutine unless they are included in the parameter list. Consequently, whatever happens in an external subroutine has no visibility outside that subroutine unless the variables involved are in the parameter list – in other words, the parameter lists make it possible for external subroutines to talk with one another.

To understand how this works, we list the code for the 'Sum and Factorial' program using **external** subroutines with parameter lists. To confirm where that the subroutines are external, note the position of the END statement.

Analysis

In order to decide on what parameters are to be passed to a given subroutine, you begin by asking, 'what information is required by the subroutine to accomplish its task'. Clearly, that is the basic information which must be included in the parameter list. In the Sum/Factorial program, it is obvious that the value of N which gleaned in the InputQuery subroutine must be passed to every sub that needs it and that includes ComputeSum, ComputeFactorial and PrintResults. In addition, PrintResults needs to know the value of Sum and Factorial in order to print these values. This means that all three variables, N, Sum and Factorial must be passed back to Main so that PrintResults will have access to these values. However, simply passing these variables back to Main will accomplish nothing unless all three variables are also included, as was done, in the parameter list for PrintResults.

In addition, we had to include the string variable Ans$ in the parameter list for RepeatQuery since the IF statement needed to know if Ans$="N" in order to act on it if 'true'.

Note that the variable Count, which is used in both ComputeSum and ComputeFactorial, is strictly local to these subroutines, and is therefore completely unknown elsewhere in the program. In this way, one can use the same variable name in different subroutines for the same or even a different purpose (a practice that is not advised) without worry about unexpected interaction. Remember 'Minimize Scope' for more hassle-free programming.

Parameters and Parameter Passing
Now that we have a feel for how parameters are passed between subroutines we will make the subject a little more formal.

Parameter Syntax
Numeric variables, string variables and expessions are included in the parameter list exactly as they are written. Arrays have a slightly different syntax. One dimensional arrays are coded as A(), two dimensional arrays as A(,), three dimensional arrays as A(,,) etc. For example we might write,

```
Sub DataInput (X(),Y(,),X$(),Count, Sum,N,Ans$,A$&B$,Z^2-2)
```

where X() and X$() are one dimensional arrays (numeric and string), Y(,) is a two dimensional array, Count, Sum and N are ordinary numeric variables, Ans$ is a string and A$&B$ and Z^2-2 are string and numeric expressions respectively.

In the CALL statement, you have the option of not including the () for the arrays, as in the statment, CALL DataInput(X,Y,X$,Count,Sum,N,Ans$,A$&B$,Z^2-2).

How Parameters are Passed
The parameter lists in the CALL statement and in the subroutine definition must match exactly as ***to both the number of parameters and the variable type,*** that is, numeric or string, but matching is not required as to the name of the variables. Thus when the variables are passed to the subroutine, parameter 1 in the CALL statement is passed to parameter 1 in the subroutine, parameter 2 in the CALL statement to parameter 2 in the subroutine, and so on. In terms of computer memory, this means that, for example, the statements,

```
CALL ComputeStuff(  3.2,  y,"No", Data)
                      ↓    ↓    ↓    ↓
SUB ComputeStuff(    x,    y, Ans$,  z())
```

are compatible since x and y are numeric variables, “No” is a string and, provided Data has been previously dimensioned as a one dimensional array, Data is compatible with z(). This example should make it clear that the variables in the SUB statement are really ‘dummy’ variables that serve to take on the values of the parameters passed in the CALL statement.

Had Data and “No” been reversed in the CALL statement or had there been a different number of arguments in the CALL statement than the SUB statement, a compile time exception would have occurred and execution would have halted.

One of the useful advantages of subroutines is that the transfer of data between the portion of the program containing the CALL statement and the external subroutine can be reversible, that is, in both directions but it need not be if one wishes to protect variables from change in the CALL routine.

To understand this, we need to differentiate between passing a parameter by value (or copy), and passing by reference (or address). Subroutines have the capability to pass by either method although the most common method is by reference, whereas functions, a topic to yet be explored, can pass parameters by value only.

Passing by Value or Copy
When an argument (variable) is passed by *value*, a copy of the value of the variable is assigned to the receiving subroutine’s parameter. This has the effect of protecting the original value of the variable in the calling routine because, while the called subroutine can alter the value of the parameter, it only alters the copy and not value of the original variable. In this way, the variable has two values, one in the calling routine and one in the called subroutine and the two are independent. This is how parameter passing works for *functions* (to be discussed shortly), but it is not the default method for subroutines which usually pass parameters by *reference*.

Passing by Reference or Address

When a variable's value is passed by *reference*, the variable's memory address (but not its value) is assigned to the called subroutine's parameter. Thus the called subroutine knows where to look to find the current value of the parameter. In this case, when the subroutine changes the value of the parameter it will store the new value at the same memory location the parameter had previously thus altering the value of the variable in both the calling routine and the called subroutine. In other words, the new value of the variable is passed back to the calling routine. Depending on what you want to do, this may or may not be desirable.

Here are some examples illustrating both kinds of parameter passing.

```
Example 1 (Passing by value)
!No information passed back to Main
!Main
CALL BlackHole(2,5)
PRINT x,y,z       !get 0,0,0
END

SUB BlackHole(x,y)
   z=x*y
  PRINT x,y,z   !get 2,5,10
END SUB
```

Here, the constants, 2 and 5 are passed to BlackHole whereupon x and y are defined as 2 and 5 respectively and z is computed. However, x, y and z remain strictly local to BlackHole and are not passed back to Main as shown by the 0 values for the variables. This method of passing parameters is of little practical value in most programming circumstances.

The most common method of passing parameters is by *address* as the following program illustrates.

```
Example 2 (Passing by address)
!x,y and z are passed back to Main
!Main
 x = 2
 y = 5
 z = 10
CALL Test(x,y,z)
PRINT x,y,z       !get 4, 25, 29
END

SUB Test(x,y,z)
   x = x^2
   y = y^2
   z = x + y
  PRINT x,y,z   ! get 4, 25, 29
END SUB
```

This example differs from the previous one in that the argument list for the CALL and SUB statements are identical to one another. Consequently, the current values of x, y and z in Main are passed to Test, changed and then passed back in altered form. This is typically how external subroutines pass parameters -- by address.

This last example shows that if the arguments of the CALL and SUB statements are identical, then whatever changes are made to the variables in the subroutine will be carried back to the calling routine (Main in this case). This may or may not be a good thing. One often chooses to protect the variables defined in the CALL routine from change while at the same time using those variables to compute some new quantity in the subroutine. In other words, we might want to protect the variables in the calling routine from changing.

Here are a couple of examples of how this might be done.

The first example takes advantage of the fact that a subroutine can alter any variable used as a parameter in a CALL statement but it cannot alter an *expression*. To make an *expression* of a numeric or string variable (but not an array) we have only to enclose it in parentheses as in (x) or (x$). So, for example, if we re-write the CALL statement in the previous example as CALL Test((x),(y),(z)) so that now x, y and z are deemed expressions, and run the program, we find that x, y and z retain the original values in Main of 2, 5 and 10 instead of 4, 25 and 29.

While the previous method works for simple variables, it will not work for arrays. To deal with this problem we can make use of the fact that the arguments in the parameter list are *dummy variables* - a fact that we will see can afford a strong measure of protection.

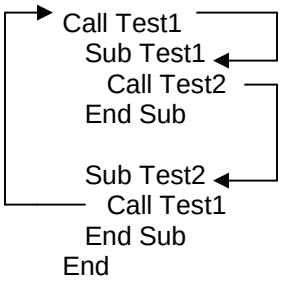
Suppose we have two one dimensional arrays, X and Y whose values we do not want to change. However, we do want to use these values to compute two new arrays, Xnew and Ynew and then use these values for other purposes. The following program illustrates how this can be done.

```
DIM x(5),y(5),Xnew(5),Ynew(5)
!First we make a couple of arrays whose values we do not want to alter! Note that Xnew and Ynew
!are lists whose five values are initially all zero since we dimensioned them but we did not assign
!them any values yet.

FOR i=1 to 5
    x(i)=i
    y(i)=i*i
NEXT i

PRINT "original x,y data"
MAT PRINT x,y                    !here's the original data

CALL MakeNewData(x,     y,     Xnew,Ynew)
PRINT
PRINT "unaltered x,y data"                    !shows that the original data is not changed
MAT PRINT x,y
PRINT
PRINT "Xnew and Ynew"                         !here's the new array
MAT PRINT Xnew,Ynew
END                                           !here's our end statement

SUB MakeNewData(xold(),yold(),Xnew(),Ynew())  !note this is an external sub and its parameter list
                                              !includes Xnew and Ynew.
   FOR i=1 to 5
      Xnew(i)=10*xold(i)
      Ynew(i)=10*yold(i)
   NEXT i
END SUB
```

Here, Xold and Yold are 'dummy' variables that take on the values of X and Y. Xnew and Ynew are then generated and passed back to Main. The important thing is that X, Y are not altered. Note that we dimensioned Xnew and Ynew in the calling routine and not in MakeNewData. ***This is because an array must be dimensioned before it is referenced***, and since Xnew and Ynew were referenced in both the MAT PRINT and in the argument list of the CALL statement, the DIM statement had to go in Main. Typically, one would dimension a variable to 1 in Main and then *redimension* later in the external subroutine once the dimensions where known. Note however that had there been no reference to Xnew or Ynew in Main, then the DIM statement for these arrays would have gone in MakeNewData. Note as well that we did not have to dimension the dummy variables Xold and Yold because they inherit the dimensions of X and Y when they are passed. An alternative way to protect X and Y is to define a couple of new variables, say Xdummy and Ydummy so that Xdummy(1)=X(1), Ydummy(1)=Y(1) and so on and then pass the Xdummy and Ydummy arrays instead of X and Y.

NESTED SUBROUTINES

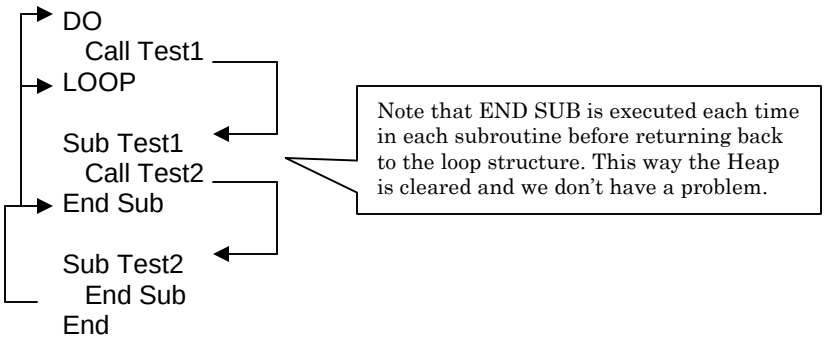
Here is a subtle problem to watch out for when using Calls from within a subroutine. Type in and run the following program.



Here Test 1 is called which does nothing more than call Test2 which in turn calls Test 1 and so on. An infinite loop you say, but in fact the program quickly crashes with an 'out of memory' error.

The reason the problem arises is that neither Test1 nor Test2 were ever allowed to finish by executing the END SUB statement in either program. When a subroutine finishes naturally, i.e., either through the END SUB or through and an EXIT SUB (which just transfers execution to the END SUB) execution is passed back to the line following the CALL statement. When this occurs, the call routine's stored variables are cleared from the Heap, which is a region in memory (@ 1MB) for variables whose lifetimes are generally unknown at the time of program execution. Once these variables are cleared, the problem is solved.

Returning to the previous program, we can re-write it with a simple DO/LOOP to ensure that each subroutine is cleared through the END SUB as follows,



This time, when Test 1 is called, Call Test2 is executed, and so is Sub Test2 but this time Test2 finishes at the End Sub. Flow then passes to the End Sub in Test1which is also cleared for the next pass when the LOOP is executed. Now you have an infinite loop and no memory problems. If you run this be sure to put a counter in the DO/LOOP with an exit otherwise you will have problem with stoping the program.

PROGRAMMING TIP

Always allow subroutines to finish either through the END SUB or force the issue with an EXIT SUB.

One way to facilitate this is to limit the number of tasks a does subroutine does (ideally to one) so that premature exits are not necessary. If one does leave a sub early, that is before the END SUB, through a call to another subroutine, then use an EXIT SUB at some point, say, on the line following the call to force a proper exit when execution returns to the calling routine.

Subroutines Calls to Other Subroutines

The calling of subroutines is very straight forward when dealing with external subs because any external subroutine can call any other external subroutine. Just make sure the parameter lists are compatible.

FUNCTIONS

Functions are like subroutines in that there are both internal and external functions, and like subroutines, they also use parameter passing. Functions differ from subroutines in that the parameter passing is only one way namely, from the call statement to the function and not back again. In fact, the only value a function can pass back to the calling routine is the value associated with the function's name. So, if you

use a function, make sure you set the name of the function equal to the computed value you want returned.

Like subroutines, external functions are external to the END statement while internal functions are internal to the END statement. And, like internal subroutines, internal functions are potential trouble makers so it is recommend that you use external functions exclusively and that means you must use a parameter list in most cases.

Declaring an External Function
When you use an *external function*, you must declare it, that is, tell the complier that the function exists somewhere after the END statement. This is done with the statement **DECLARE DEF** *Function Name*

where this statement occurs somewhere before the function is invoked and before the END statement. In addition, functions can be either single line or multiline. In either case, one must define the function with the keyword, DEF.

Defining a Function

```
DEF RollDie=Int(6*RND+1)          !single line definition

DEF Response(Par(),X(),Y())       !multi line definition
    For i=1 to NumObs
        Yobs=Y(i)
        Ycomp=Par(3)*X(i)^3+Par(2)*X(i)^2+Par(1)*x+Par(0)
        Residual=Yobs-Ycomp
        Sum=Sum+Residual^2
        Response=Sum              !associate the name of the function with a computed value
    Next i
END DEF
```

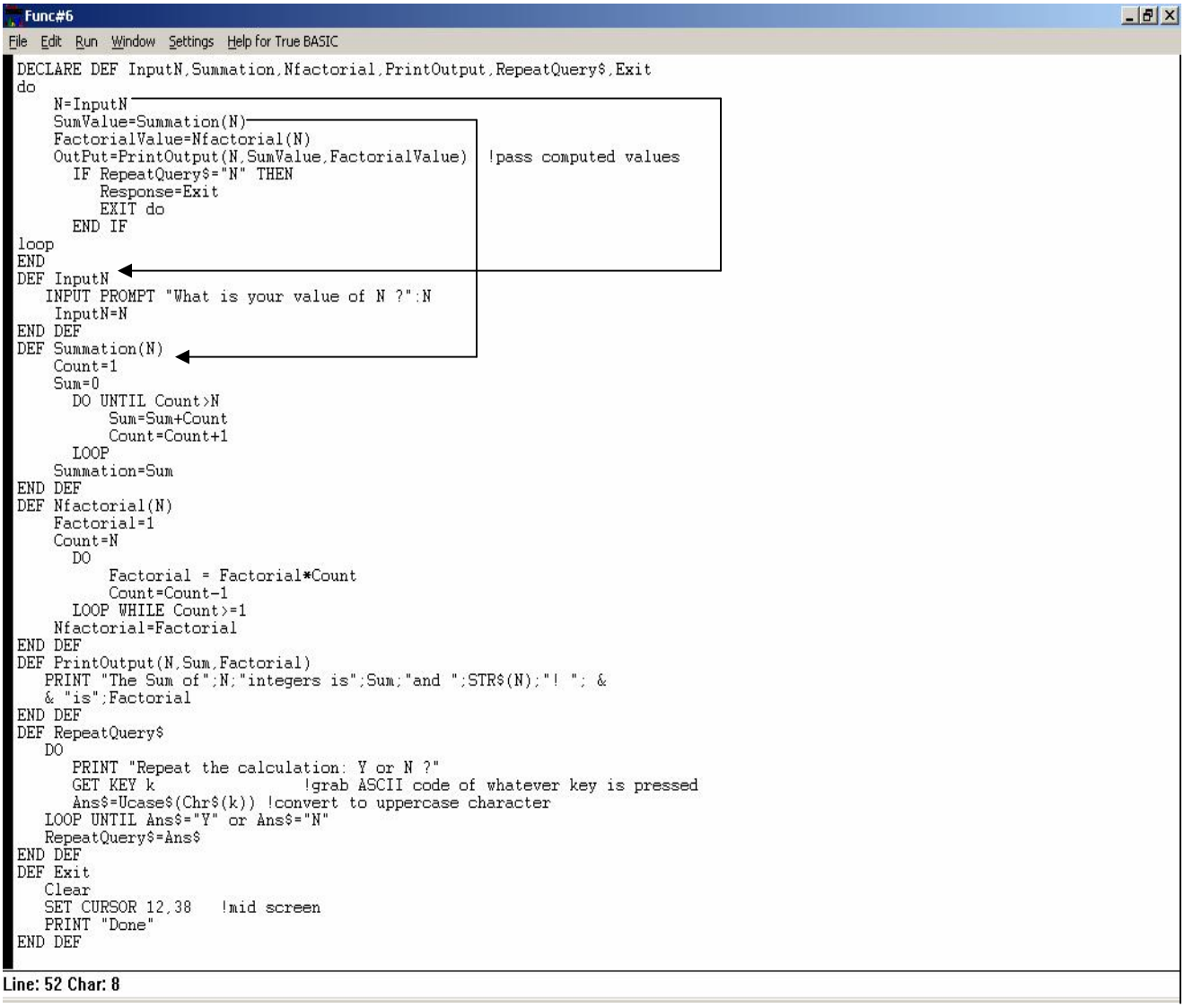
In either case, we have to use the DEF keyword but for the multiline case we also have to tell the complier when we are through defining the function with the keyword END DEF.

Invoking a Function
Invoking a function is simple, just use its name, as in

```
Result=RollDie                    !single line case
FinalSum=Response(Par,X,Y)        !multi line case
```

Note that the value computed as a result of invoking the function is returned as the function name itself i.e., in the multiline case, the name of the function, Response is set equal to the computed result Sum.

The following screen shot is of our old friend the Sum and Factorial of N Integers program written using external functions calls instead of subroutines.



Note all of the functions declared on a single line (to make the screen shot possible) and the location of the END statement. Clearly, these functions are all external.

GRAPHICS

One of the strong points of True Basic is the power and ease of use of its graphic commands. The following is an outline of the commands available.

The Output Window in which any graphical object will be displayed has a default coordinate system ranging from 0 to 1 along both axes. This means that if you want to plot either points or draw lines whose coordinates fall between 0 and 1, the default window will work just fine. For any other coordinate range you will need the statement:

SET WINDOW Xmin, Xmax, Ymin, Ymax
Here, Xmin etc. define the axes range in both directions. Thus if you are plotting data whose X axis ranges from -23.6 to 123.9 you might set Xmin to -25 and Xmax to 125. The same is true for the Y axis.

Plotting Points and Lines
a) **Points**:

PLOT POINTS as in: **PLOT POINTS:** x,y – produces a dot on the screen at the coordinates, x,y.

Example:
Here is how you would plot a sine curve on the screen.

```
SET WINDOW 0,2*PI,-1.1,1.1
  For x=0 to 2*PI Step 0.05      !the smaller the increment, the more the points will resemble a line
    PLOT POINTS: x,sin(x)        !or use y=sin(x) and Plot Points:x,y
  Next x
End
```

Alternatively, you could have used a Read/Data statement had you had data you wanted to plot, as in:

```
Do While More Data
  Read x,y
  Plot Points x,y
Loop
Data x1,y1,x2,y2,….etc.
End
```

b) **Lines**:

PLOT LINES:x1,y1;x2,y2
This makes sense if we remember that the delimiter ';' suppresses the carriage return so Plot Lines:x1,y1;x2,y2 simply plots two points but does not lift the pen between them thus connecting them with a line.

Note: when plotting multiple unconnected lines on the same display or printer page, you must insert a **PLOT** statement immediately following the code that draws each successive graph. This makes sense if you think of the drawing process as involving a pen that moves over the paper (or screen) and in so doing, draws a line. As we have seen, the ';' or semicolon acts to keep the pen 'down'. **PLOT** serves to lift the pen so you can draw new line without a line connecting the last point of the previous line to the first point of the new line. Sounds complicated, but it's easy to use in True Basic.

c) **Areas**:
PLOT AREA X1,Y1;X2,Y2;…;Xn,Yn
Plot the points, connects a line between them including between Xn,Yn and X1 and Y1, and then fills the area with the color of the line itself.

d) **Text**:
PLOT TEXT, AT x,y: s$
x and y define the beginning of the text s$ which has to be a string, i.e., something in quotes.

Here is how you would label the X axis using the Plot Text statement.

```
For x=1 to 10                    !x is a numeric variable
  X$=STR$(x)                     !convert the number x into a string variable
  PLOT TEXT, AT x,-0.04:X$       !now plot X$ slightly below the X axis
Next x
```

e) **Graph Labels**:
SET CURSOR row,col

This command will set the cursor at the specified row and col after which you may print something. However, the maximum column and maximum row depends on the monitor you have so you need to interrogate the computer for its maximum values of each of these constants. To do this you can write:

ASK MAX CUSOR maxrow,maxcol

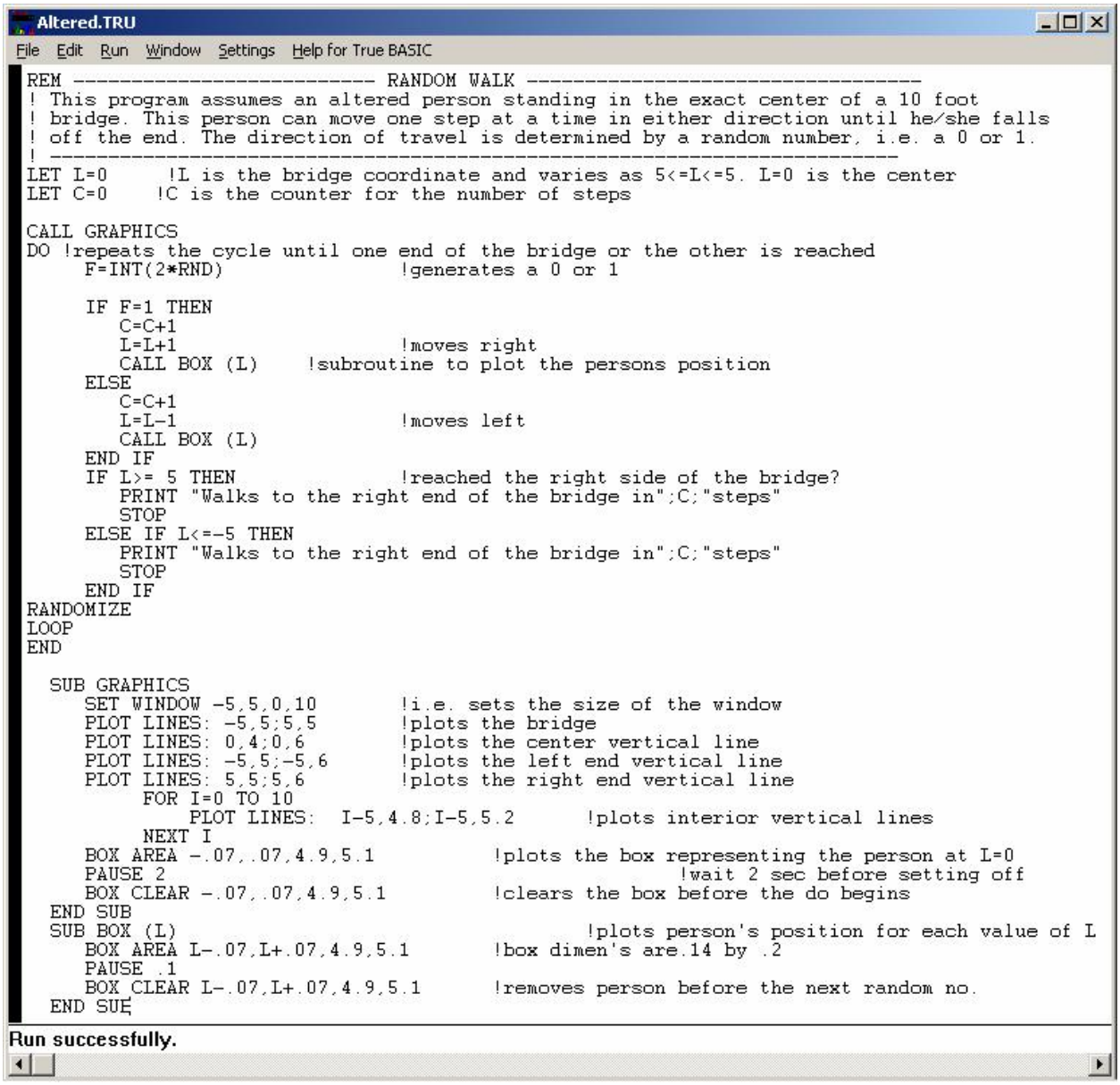
Maxrow and maxcol will now be the point at the extreme bottom right of the monitor. Using these two values you can now set the cursor the **SET CURSOR** statement without worrying about getting a 'cursor out of bounds' error.

Other Commands include:
BOX AREA, **BOX CIRCLE**, **BOX CLEAR**, **FLOOD** etc. See your text for descriptions.

Examples:

1) One Dimensional Random Walk Problem (classic problem sometimes called the ‘drunkard’s walk’)

Consider a person who has enjoyed himself/herself a little too much and in so doing become slightly ‘altered’. Suppose that person is standing on a one dimensional walkway holding onto a lightpost. This person can now take a fixed length step to the left or right but each step is completely ‘random’ i.e., without forethought. The walkway is of length five steps in either direction after which he/she will fall off the walkway. The problem is to model this ‘one dimensional random walk’ problem on the computer. The following program does just that through the use of the random number generator.



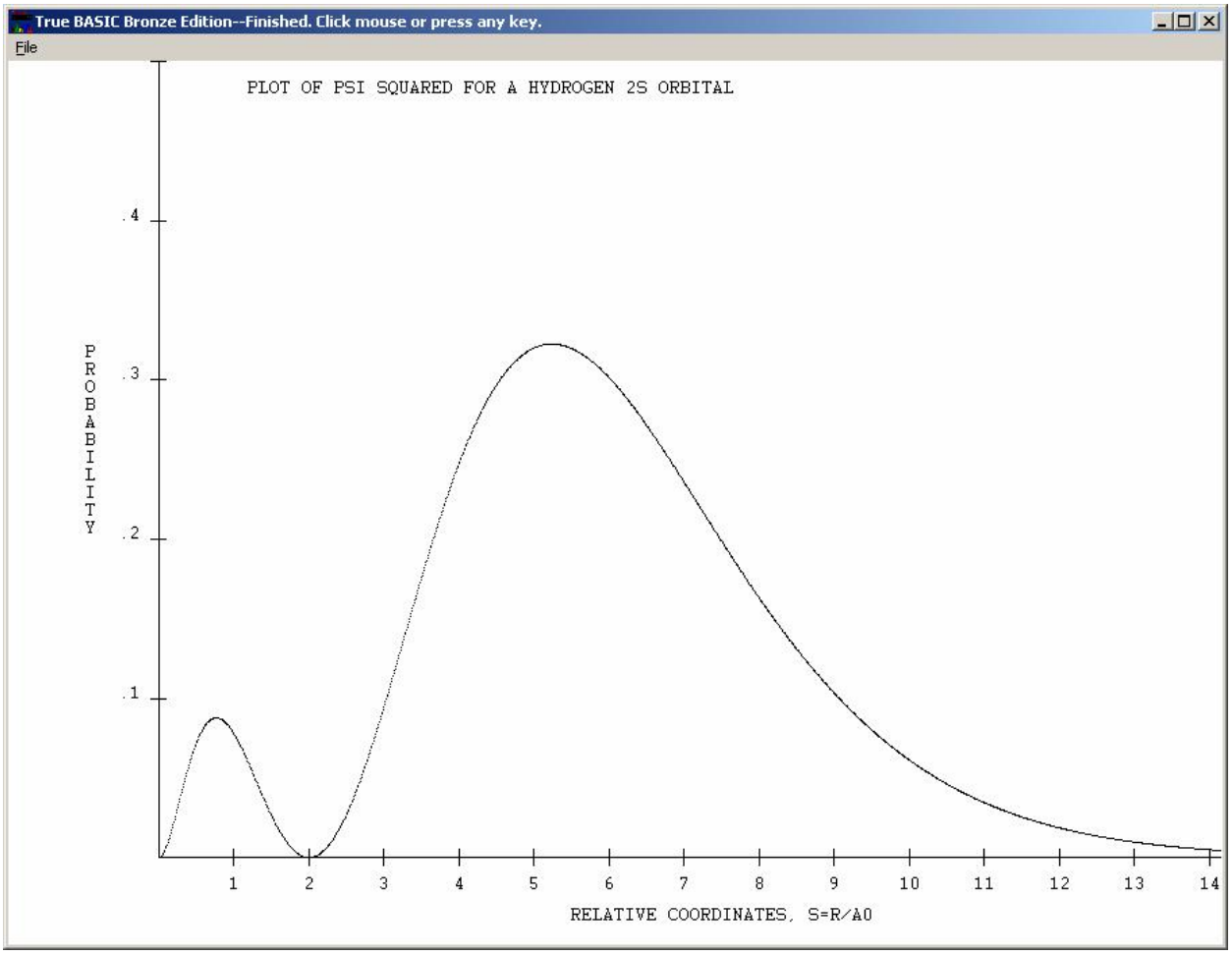
You can find the program in the list under ‘2DRandomWalk.tru’ and run it to see how it works.

2) Plot of a 2-S Hydrogen Orbital

This program illustrates the various steps involved in ploting a function, setting up the axes with labels etc.

```
!---------------------- PLOT OF A 2S HYDROGEN ORBITAL ----------------------
! Windows Version, Internal Subroutines

CALL AXES                       !sets the window and draws the axes
CALL TICKS                      !puts in the tick marks
CALL AXES_LABELS                !adds the labels

FOR S=0 TO 15 STEP 0.01         !screen printing routine
   CALL PLOT                    !plots the function
NEXT S

SUB AXES
   SET WINDOW -2,15,-.05,.5     !X from -2 to 15; Y from - 0.05 to 0.5
   PLOT LINES: 0,0;15,0         !draws x axis
   PLOT LINES: 0,0;0,.5         !draws y axis
END SUB

SUB TICKS
   FOR X=1 TO 15 STEP 1
      PLOT LINES: X,-0.005;X,0.005
   NEXT X
   FOR Y=0.1 TO 0.5 STEP 0.1
      PLOT LINES: -.1,Y;.1,Y
   NEXT Y
END SUB

SUB AXES_LABELS
   FOR X=1 TO 9                 !split X numbering into 2 parts to get spacing right
      X$=STR$(X)
      PLOT TEXT, AT X,-0.02:X$
   NEXT X

   FOR X=10 TO 14               !here's part 2
      X$=STR$(X)
      PLOT TEXT, AT X,-0.02:X$
   NEXT X

   FOR Y=0.1 TO 0.5 STEP 0.1    !Y axis numbering
      Y$=STR$(Y)
      PLOT TEXT, AT -0.4,Y:Y$
   NEXT Y

   SET TEXT JUSTIFY "CENTER","BASE"   !add labels - start with X axis label
   PLOT TEXT, AT 7.5,-0.04:"RELATIVE COORDINATES, S=R/A0"
   A$="PROBABILITY"             !Y axis label
   L=LEN(A$)                    !length of A$
   SET CURSOR 25,1              !starts cursor at row 25, column 1

   FOR I=1 TO L
      PRINT TAB(8); A$[I:I]     !moves to col. 8 and prints a ter, then
   NEXT I                       !carriage returns to column 1, next row etc.

   SET CURSOR 2,23              !graph's label
   PRINT "PLOT OF PSI SQUARED FOR A HYDROGEN 2S ORBITAL"
END SUB

SUB PLOT                        !plot the prob function for the 2S H atom orbit
   A0=0.5921                    !these 4 statements define the H atom 2 S orbital
   C=4*PI*(S*A0)^2
   PSI=(1/(4*SQR(2*PI)))*((1/A0)^(3/2))*(2-S)*EXP(-S/2)
   PROB=C*PSI^2
   PLOT POINTS: S,PROB          !Plotting statement
END SUB
END
```

The following output window shows the results of running the program. Note that we are ploting points, not a line but if the step size is small enough, there is very little difference from the way the plot looks. The program is stored under 2SHPlotWinVer.Tru.



Analysis:
You will note that the program consistes of three subroutine calls (Axes, Ticks, Axes_Labels) that set up the graph itself, followed by a call to 'Plot' that plots the graph. For the most part, the comments are pretty self explanatory. The only part that might not be obvious is the code fragment:

```
A$="PROBABILITY"          !Y axis label
L=LEN(A$)                 !length of A$
SET CURSOR 17,1           !starts cursor at row 5, column 1
FOR I=1 TO L
   PRINT TAB(8); A$[I:I]  !moves to col. 8 and prints a letter, then
NEXT I                    !carriage returns to column 1, next row etc.
```

Here A$ is the string we want to print. LEN(A$) computes the length in characters of A$ which in this case is 11 (i.e., 11 characters in the word 'PROBABILITY'). The SET CURSOR statement just defines the starting point for printing. A$[i:j] is a very useful string operator which functions to identify the substring contained within A$ starting with the i^{th} character and ending with the j^{th}. Thus, if i=1 and j=11 then the substring is identical to A$. Now consider the For/Next loop and note that j=i in the PRINT TAB(8); A$[i:i] statement. By setting i=j we are successively printing every character belonging to A$ as we move through the loop. The TAB statement ensures that we move down one row after each character is printed. Thus we get A$ printed vertically on the Y axis. True Basic has no method to rotate the letters to the left by 90^o (and print from bottom up as is standard in journals) so this is the best we can do. Still, it works fine for our purposes.

FILES

True Basic support five files type: Text (or Sequential), Random, Record, Stream and Byte. We will only discuss two of these, namely, Text and Random file types. Of these two, we recommend using Text files exclusively because they have the advantage of producing data in a form that can be imported into Excel or Word since they recognize the ASCII format. This is likely to be an issue when one is using True Basic to communicate with an instrument that generates a lot of data or when one writes a True Basic program that computes data that can be best analyzed in a Spread Sheet format. The downside of Text files is that data can only be read from start to finish and new data can only be added to the end of the file. This kind of file would be of little use to a business, for example, who had a large alphabetized data set where searching would be made easier if each data record could be accessed when needed and new data could be inserted in alphabetical order.

General Comments:
All file types have certain procedures in common even if the exact syntax will differ with the file type.

Opening a File

a) Text Files: **OPEN** channel #: **NAME** filename$, **ACCESS** "OUTIN", **CREATE** "NEWOLD", **ORG** "TEXT"
b) Record Files: **OPEN** channel #: **NAME** filename$, **ACCESS** "OUTIN", **CREATE** "NEWOLD", **ORG** "RECORD"

Comments:
Channel #: an integer of your choice between 0 (reserved for the screen) and 999. In the example below we arbitrarly chose Channel #1 for the file we were opening and writing to and Channel #2 for the printer. Your choice but you must be consistent within a given program.

Name: filename$ is a string and includes the path to where you want the file written, i.e., to a hard drive as in "C:\trash\gaslawdata" which assumes a directory 'trash' exists etc.

ACCESS, **CREATE**, **ORG** are all required keywords and the phrases 'outin', 'newold' are the default and most flexible methods of writing and reading existing or creating files. **ORG** simply specifies what kind of file you are working with.

Other Commands

CLOSE *channel* # - closes a file – always close the file immediately after opening or writing to it!
ERASE *channel* # - erases the file's contents
UNSAVE *channel* # - removes the entire file – make sure the file is closed before unsaving it!
OPEN channel #: **PRINTER** – opens the printer channel
PRINT *channel* #: [variable list in a loop of some sort] - sends output directly to the printer

Writing/Reading Data
Again, we have to distinguish between Text and Record files. In the examples below, we are assuming a FOR/NEXT loop to either read (READ/DATA) the data in or input it from the keyboard.

	Writing Data to a File	Reading Data in a File	Comments
Text File:	**PRINT** #1: X(I); ","; Y(I)	**INPUT** #1:X(I),Y(I)	note the ","; syntax for 2 dim data
Record File:	**WRITE** #1: X(I),Y(I)	**READ** #1: X(I),Y(I)	record files require additional commands – see the example

Both of the following examples are available in the machines in the computer laboratory. You should run them to get a feel for how they work.

TEXTFILE.TRU

File Edit Run Window Settings Help for True BASIC

```
!------------------------------ TEXT FILE EXAMPLE ------------------------------
!Program to demonstrate how to open a text file, write to it,
!close it, open it again and print the results to the printer.
!-------------------------------------------------------------------------------
DIM X(3),Y(3)
!note that the NAME requires the full path name -
OPEN #1: NAME "A:\TextFileExample", ACCESS "OUTIN", CREATE "NEWOLD", ORG "TEXT"
ERASE #1                                    !start by erasing the file

FOR I=1 TO 3
    PRINT "INPUT X(";I;")";"AND Y(";I;")"   !screen prompt - e.g., 1,2 etc
    INPUT X(I),Y(I)                         !input the data to memory
    PRINT #1: X(I); ","; Y(I)               !input data to the file, the "," is required!
NEXT I

CLOSE #1  !file is saved in the directory C:\trash\ named Textfile$

PRINT "Press any key to print to the printer."
GET KEY Q                                   !stops execution until a key is pressed

! Now let's open the file and print the results to channel #2

OPEN #1: NAME "A:\TextFileExample", ACCESS "OUTIN", CREATE "NEWOLD", ORG "TEXT"
OPEN #2:PRINTER  !direct output to the printer
PRINT #2: "Here is the data in the file."        'prints header for the data

FOR I=1 TO 3
   INPUT #1: X(I),Y(I)                           !reads the file
   PRINT #2:"X(";I;")=";X(I);"  Y(";I;")=";Y(I)  !sends output to the printer
NEXT I

CLOSE #1  !close data file channel
CLOSE #2  !close printer channel
END
```

Saved.

RECORDFILE.TRU

File Edit Run Window Settings Help for True BASIC

```
!------------------------------ RECORD FILE EXAMPLE ------------------------
!Program to demonstrate how to open a record file, write to it,
!close it, open it again and print the results to the printer.
!-------------------------------------------------------------------------------
DIM X(3),Y(3)
OPEN #1: NAME "A:\TextFileExample", ACCESS "OUTIN", CREATE "NEWOLD", ORG "RECORD"
ERASE #1 !start by erasing the file
SET #1: RECSIZE 8   !tells the computer to use 8 bytes for floating pt. no's.

FOR I=1 TO 3
    PRINT "INPUT X(";I;")";"AND Y(";I;")"   !screen prompt
    INPUT X(I),Y(I)                         !input the data to memory
    WRITE #1: X(I),Y(I)  !note how this differs from the text file
NEXT I

CLOSE #1  !file is saved as TextFileExample on a floppy disk

PRINT "Press any key to print to the printer."
GET KEY Q                                   !stops execution until a key is pressed

! Now let's open the file and print the results to the printer channel #2

OPEN #1: NAME "A:\TextFileExample", ACCESS "OUTIN", CREATE "NEWOLD", ORG "RECORD"
OPEN #2:PRINTER  !direct output to the printer
PRINT #2: "Here is your data in the file"  !print header
SET #1: POINTER BEGIN  !need to reset the pointer for record files

FOR I=1 TO 3
   READ #1: X(I),Y(I)   !note that we use READ not INPUT
   PRINT #2:"X(";I;")=";X(I);"  Y(";I;")=";Y(I)
NEXT I

CLOSE #1  !close data file channel
CLOSE #2  !close printer channel
END
```

Line: 1 Char: 1

These programs are stored on the computers under TextFile.Tru and RecordFile.Tru.