# VCU Bioinformatics and Bioengineering Summer Institute

**I**ntroduction to Personal Programming with True Basic (6 June 2003)

I. Why program?

II. Overview of True Basic syntax

A. Sample program in True Basic
B. Variables
C. Operators
D. IF blocks
E. Loops
F. Subroutines and functions

III. Structured programming

A. The need for structured programming
B. The process of writing a structured program

## I. Why program?

Here's a typical situation in bioinformatics

You are examining the 967-nucleotide DNA sequence upstream from the gene that you work on, and you happen to notice something in it that astonishes you. You were just reading about the mechanism by which a protein, NtcA, binds to DNA and thereby affects gene expression, and you have a vague recollection of a table from that paper listing the proven DNA sites to which it binds (Table 1). You recall the first three conserved nucleotides GTA (stuck in your mind because it reminded you of "guitar") and the next three TAC (inverse complement of GTA). What astonishes you is that staring you in the face is this very sequence, right in front of your favorite gene! There it is, GTA and TAC separated by the required 8 nucleotides, and further downstream (between 20 and 24 nucleotides), the sequence TA . . . T (Figure 1).

Could it be that you have stumbled upon the mechanism by which your gene is regulated? You're not accustomed to the gods smiling on you in this way, and you're suspicious. Maybe this sort of sequence pops up frequently just by chance and is not by itself good evidence for the binding of NtcA. How can you tell whether celebrate your good luck or rather push away this red herring and get back to work?

After some thought, you choose to let statistical considerations decide the issue. If this sequence related to NtcA binding is very unlikely to occur in a random 967-nucleotide sequence, then you're prepared to swallow your suspicions and rejoice. How can you calculate whether this is the case? You have a nagging feeling that this kind of calculation is something you learned how to do in high school math, but who remembers those formulas when you need them? You could (a) try to remember what you never really knew (fruitless), (b) try to find something on the web (searching for what?), or (c) try to find someone who knows (at 3 am in the morning?), but the most straightforward solution is to forget math and write a simple computer program that will simulate the problem and provide an answer.

Of course, this is the most straightforward solution only if ***personal programming*** is part of the mental tools at your disposal. I want to draw a distinction between personal programming and professional programming. Professional programmers make a lot of money (and deserve it) not

## Table 1: Proven DNA sites to which NtcA binds[a]

| Strain[b] | Gene | NtcA binding site[c] | Promoter[d] |
|---|---|---|---|
| PCC 7942 | *nir* operon | AAAGTT**GTA**GTTTCTGT**TAC**CAATTGCGAATCGAGAACTGCC | **TA**ATC**T**GCCGAg |
| | n*irB-ntcB* | TTTTTA**GTA**GCAATTGC**TAC**AAGCCTTGACTCTGAAGCCCGC | **T**T**A**GG**T**GGAGCCATa |
| | *ntcA* | GAAAAA**GTA**GCAGTTGC**TAC**AAGCAGCAGCTAGGCTAGGCCG | **TA**CGG**T**AACGa |
| | *glnB* | TTGCT**GTA**GCAGTAAC**TAC**AACTGTGGTCTAGTCAGCGGTGT | **TA**CCA**A**AGAGTc |
| | *glnA* | TTTTAT**GTA**TCAGCTGT**TAC**AAAAGTGCCGTTTCGGGCTACC | **TA**GGA**T**GAAAGc |
| | *amt1* | CGAACT**GT**T**T**ACATCGAT**TAC**AAAACAACCTTGAGTCTCGCTG | **A**A**T**GC**T**TACAGAGa |
| PCC 7120 | *glnA* (I) | CGTTCT**GTA**ACAAAGAC**TAC**AAAACTGTCTAATGTTTAGAATC | **TA**CGA**T**ATTTCa |
| | *nir* | AATTTT**GTA**GCTACTTA**TAC**TATTTTACCTGAGATCCCGACA | **TA**ACC**T**TAGAAGt |
| | *urt* | AATTTA**GTA**TCAAAAAT**A**A**C**AATTCAATGGTTAAATATCAAAC | **TA**ATA**T**CACAAt |
| | *ntcB* | AAAGCT**GTA**ACAAAATC**TAC**CAAATTGGGGAGCAAAATCAGC | **TA**ACT**T**AATTGAAa |
| | *devBCA* | TCATTT**GTA**CAGTCTGT**TAC**CTTTACCTGAAACAGATGAATG | **TA**GAA**T**TTATa |
| PCC 6803 | *amt1* | TGAAAA**GTA**GTAAATCA**TAC**AGAAAACAATCATGTAAAAA | **T**T**G**AA**T**ACTCTaa |
| | *glnA* | AAAATG**GTA**GCGAAAAA**TAC**ATTTTCTAACTACTTGACTCTT | **TA**CGA**T**GGATAGTcg |
| | *glnB* | CAAACG**GTA**CTGATTTT**TAC**AAAAAAACTTTTGGAGAACATGT | **TA**AAA**G**TGTCTgg |
| | *icd* | AATTTC**GTA**ACAGCCAA**T**G**C**AATCAGAGCCTCCAGAAAGGAT | **TA**TGA**T**CTGCTCCg |
| | *rpoD2*-V | AAGTTT**GTA**TCACGAAT**TAC**ACTGCCGTGAAAATTTAACGA | **TA**TTT**T**GGACAg |
| PCC 7601 | *glnA* (P1) | GAATCT**GTA**ACAAAGAC**TAC**AAAAATTCTTAATGTCATATCCT | **TA**GGA**T**ATTCCAGgt |
| PCC 6903 | *glnN* | TTTTTT**GT**G**C**GCGTTTA**TAC**CAATCAAGTGCGATCTAATCGG | **TA**TCT**T**TTTTATc |
| PCC 7002 | *nrtP* | TAAAGA**GTA**TCAGCGGT**TAC**GAATTTAGCGAAGAAAGAATGTGA | **T**T**C**TT**T**ATCACa |
| WH 7803 | *ntcA* | GGAACC**GT**G**T**GCGTTGC**TAC**AGGGTGGGAATCGATCGCTCCT | **TA**ATT**T**CCTTGAa |
| **Consensus** | | **GTG** (8) **TAC** (20-24) | **TA**(3)**T** |

[a] Derived from Herrero et al (2001). Journal of Bacteriology

[b] Cyanobacterial strain designation.

[c] Bold face highlights highly conserved nucleotides within the region experimentally shown to bind protein NtcA. Letters in red deviate from the consensus site.

[d] Bold face highlights highly conserved nucleotides within the region thought to bind RNA polymerase. Letters in red deviate from the consensus site.

```
GAACAGCTTGAACAAAGAAAAAATGGTTGAATTAACTAAATTACAAGCAA
ATTTAGCCAAAAATTAACTCTGTGTATTCTTTTGCTCATGACGTTGGGTT
ACATTGCTAGTTTTTAAAACGTAAGTCATTTAGCTAAAAGAACTTTTAAT
AACTATAACAAAAATTTAATAATATTATCAACTTCGCTCTGGACAAGGCA
TAAACTCAACATTTTGCCAACATAGGTTATAAAAAAACGTAGAGGTAATT
GTGGCTAGAGTAACAAAGACTACAAAACCTTGGGCATGGGCTTGTTACTT
TGAAATTCATCGACGCTAAGGGGTCTTGCCGCCGTGGGTTCGGTTTGTAT
TGAAATCGTTGAGGGGAATCCCCATCTGAGGTCGTTGCTGGGTTGGCACT
TGCAACAATTGGAATACCGTGTGCATCAAGCCGCCAGCATATATCAAGCA
AGGGAAGCCTTTTTGAGCCATCAGCCAACTCTAGTGATTCTGGATGCTGA
TTTGCCAGATGGTGACGGTATTGAATTTTGCCGTTGGCTGCATCGTCAGC
```

**Figure 1.** Upstream region and partial sequence of gene *all4312* from the cyanobacterium *Anabaena* PCC 7120. The gene is shown shaded in cyano. Conserved elements of the putative NtcA binding site is shown shaded in red.

because programming is difficult. Far from it. They earn their keep because they write programs that run fast, that accommodate users with no programming experience, that work in multiple computer environments, and that anticipate a wide variety of potential disasters, all so that the program can live on its own. In contrast, a program written just for your own purposes may work only on your computer, may require your active intervention, and may work clunky... but gets the job done. If you don't know this already, you will soon realize that personal programming is easy, certainly easier than recalling high school math.

There is no single more powerful tool in bioinformatics than personal programming. Throughout the summer, you will have the opportunity of picking up a computer language that you can use in personal programming. Many of you already know one or more languages, but so that we can communicate amongst each other, the Research Simulations will make use of one easily comprehended computer language: True Basic.

**SQ1. Do you understand the scientific dilemma posed by this scenario?**

## II. Overview of True Basic syntax

Some of you are very experienced programmers, while others have little or no programming experience. To save time for the former group without depriving the latter group of what they need, I'm going to provide boxes (red text) for those who know some programming language. These may enable some of you to skip sections of text.

For the rest of you, my personal opinion is that no amount of explanation will help. What WILL help is you trying things out at the computer until the concepts make sense. You should feel free to fool around with the language in any way that occurs to you. ***There is nothing you can do short of physical violence that will hurt your computer.***

It is difficult for me to assess how much material is too much for one sitting. For those who are new to programming, what follows is surely way too much. Given the range of experience in this group, I must rely on your judgment and ask you to draw the line between what is challenging and what is overwhelming. ***Do what you can, and don't kill yourself in the attempt.***

## II.A. Sample program in True Basic[1]

First, let's examine a working program in True Basic, one that does something potentially useful. Regardless of your programming experience, all of you should be able to make sense out of a True Basic program that tries to make itself comprehensible. Download the file *Simulated_Site_Search.Tru*, and run it within your copy of True Basic. Take a look at the program by reading the brief description of the program in the first few lines and then scrolling down to the section entitled "Main Program".

**SQ2. Were you able to run the program?**

[1] These notes presume that you have True Basic loaded on your computer. Disks with the language and other material will be distributed Tuesday morning, June 3. All programs listed in these notes can be downloaded from the course web site, so instructions to "enter the program" can conveniently be fulfilled by copying and pasting from the available file.

**SQ3. Do you understand the significance of the output generated by the program in relation to the scenario described on the first page of these notes?**

**SQ4. Do you understand from the Main Program how (in principle) the output is generated?**

The Main Program should give you an outline of what the program is trying to do and how it is trying to do it. The program should be as self-explanatory as possible through the use of descriptive names and clear syntax. Comments (lines beginning with exclamation points) should provide the big picture and fill in gaps when necessary. Notice the three most critical lines of the program:

```
CALL Make_up_random_sequence
CALL Examine_sequence_for_site
IF site_found = true THEN LET occurrences = occurrences + 1
```

Perhaps these lines encapsulate the logic you might follow if you were to determine the frequency of NtcA sites yourself. They may strike you as strange, however: obviously your computer was not manufactured with the built-in knowledge of how to make random sequences nor how to examine sequences for NtcA sites. Of course, I had to teach the computer how to do these things.

If you scroll down further into the section entitled "Subroutines and Functions", you'll see how I told the program to make up random sequences. You may be able to glean from the lines within the subroutine `Make_up_random_sequence` that the program goes through a sequence of a specified length, one position at a time, and assigns one of four possible nucleotides at random. This is probably what you would do too if you had a four-sided coin and a lot of time on your hands.

You couldn't write this program in True Basic without prior knowledge of the rules of the language, but even with no experience, you should be able to get the basic idea of what a well-written program is trying to do. Furthermore, you should be able to ***modify*** such a program in simple ways. You will be doing this throughout the summer – modifying working programs to understand how important bioinformatic or bioengineering tools work.

**SQ5. What routine in the program do you think generated the lines that appeared on the screen when you ran the program?**

**SQ6. How do you think you could change the program so that instead of displaying**

***0.27 = fraction of random sequences of length...***

**the program instead displays**

***0.27 = fraction of semirandom sequences of length...***

**SQ7. How do you think you could change the program so that instead of displaying the fraction of random sequences in which sites were found, it instead displays the number of random sequences in which sites were found?**

## II.B. Variables

Variables may be thought of as boxes into which the program throws values. Thinking of them in this way avoids certain common misconceptions. For example, the True Basic statement:

```
LET x = 3
```

does not ***equate*** the variable x with the number three. Rather it puts in a box called x the value 3. It is perfectly OK to say:

```
LET x = 3
LET x = 4
```

No contradiction here. The program simply writes the number 3 into the box called x and then writes the number 4 into that box (erasing the previous value).

- True Basic has only two types of variables: strings of unlimited length and 8-byte floating point numbers.
- Variables need not be declared.
- String variables are recognized by names (up to 31 characters) ending with the "$" character (e.g. sequence$).
- Numeric variables are those with names (of any length) that don't end with "$".
- Special functions are available to do higher precision math and to manipulate strings as bit strings.
- Scalar assignments have the syntax: LET *variable = value*
- "=" has the function of both assign and logical compare (for both strings and numbers), depending on context.

**SQ8. Predict what will appear if you run the following program. Then enter and run it.**

```
LET x = 3
LET y = 4
LET x = y
LET y = 5
PRINT x
END
```

Like most languages, True Basic makes a distinction between variables that contain numbers and those that contain text. Numeric variables can contain numbers like 10, -3.14, 6.02e23 (interpreted as $6.02 \cdot 10^{23}$). String variables (those that contain text) can hold values like "armadillo", "ten", or "10" (the symbols "1" and "0", not the number 10).

The names of the variables indicate what it can hold. Names that end in "$" hold text, not numbers. Names that ***don't*** end in "$" hold numbers, not text. Names can be as long as you like, so long as they don't contain blanks.

**SQ9. Predict which of the following statements True Basic will complain about (because they're syntactically incorrect). Then enter and run the program to find out. Correct the offending statements so that they're OK.**

```
LET necessity$ = "mother of invention"
LET one = 2
LET two$ = 2
LET three$ = "2 + 2"
LET four$ = two$ + two$
PRINT necessity$, one, two$, three$
END
```

## II.C. Operators

For the most part, numbers and numeric variables in True Basic are acted on by operators that you're familiar with and parentheses work as you'd expect.

- Arithmetic operators: + - * / ^
- Comparisons (strings and numbers): = > < >= <= <>
- Logical operators: AND, OR, NOT
- String operator: & (concatenate)
- All types of brackets are inter-changeable: ( ) [ ] { }

**SQ10: Predict the output of the following program. Then enter and run it to find out the real story. Fool around with the numbers until you're satisfied you understand what each symbol means.**

```
PRINT "+ and - operators: "; 5 + 2 - 3
PRINT "* and / operators and parentheses: "; 2*(3+4)/6, 2*3+4/6
PRINT "^: "; 1 + 2^3
END
```

Both numbers and strings can be compared to each other in (mostly) expected ways:

```
IF angle = 2*pi THEN...
IF fragment_size > resolution_of_gel THEN...
IF restriction_site$ <> "GAATTC" THEN...    ! (<> means not equal)
IF dictionary_entry1$ > dictionary_entry2$ THEN...
```

Strings can be compared, as shown in the last example, where "greater than" generally means "later in alphabetical order".

**SQ11: Play with the following program, replacing the numbers with other numbers or with strings and changing the comparison operators, until you're convinced you know how each symbol works.**

```
! Symbols to choose from: =   >   <   >=   <=   <>

IF 1 = 2 THEN PRINT "yes" ELSE PRINT "no"
IF "truth" > "Beauty" THEN PRINT "yes" ELSE PRINT "no"
END
```

Programming in bioinformatics focuses more than most types of programming on long strings, i.e. DNA or protein sequences. True Basic provides a number of ways to access and manipulate strings. The most fundamental is the extraction of an internal portion of a string. For example, if you had stored in genome$ the entire 6 million-nucleotide DNA sequence of a bacterium's genome and wanted to pull out of it a specific gene (whose coordinates you know) you might write something like:

```
LET start = 4352066
LET end = 4353115
LET gene$ = genome$[start:end]
```

- Substrings are numbered from 1,not 0
- Substrings are accessed:
  `string$[begin:end]`
  where `begin` and `end` are the termini of the substring. (If `begin` < 1, you get the left portion, if `end` > length of string, you get the right portion.
- Substrings are assigned:
  `LET string$[begin:end] = a$`
  If `begin` > `end`, then the contents of a$ are inserted with nothing lost.
- Many string functions are available (described later).

```
! Coordinates of extents of gene
! Coordinate of end of gene
```

A similar syntax is used for altering strings.

**SQ12: Play with the following program, predicting each time what will be the output. Try replacing the numbers within the brackets with other numbers, as strange as you can imagine (huge numbers, fractions, negative numbers, transcendental numbers...) until you have deduced the rules that govern substrings.**

```
LET string$ = "Give me liberty or give me death"
LET word$ = string$[9:15]
PRINT word$
LET string$[20:19] = "maybe "
PRINT string$
END
```

**SQ13: Modify the above program so that the printed phrase appears to pose a choice between freedom and food. The only changes you are allowed to make are those of the form:**

```
LET ... = ""          ! Put nothing between the quotes
```

### II.D. IF blocks

- `IF` *condition* `THEN`
  *statements*
  `ELSE IF` *condition* `THEN` `!*`
  *statements*
  `ELSE` `!*`
  *statements*
  `END IF`
- Statements marked "!*" are optional

```
!* These statements optional
```

What distinguishes a computer from a calculator? Nowadays, not much, but in principle, the defining characteristic of a computer is the ability to store instructions and reuse them in an order that may be determined at the time of execution.

True Basic tries to mimic your internal deliberations: `IF` something is true `THEN` do the following things. Otherwise (`ELSE`) do something else. You can see this syntax in action in the site search program you examined at the beginning of these notes:

```
IF subsequence$[12:14] = "TAC" THEN
   LET site_found = true
ELSE
   LET site_found = false
END IF
```

An idiosyncrasy of the language is that you can write this logical sequence in only one of two ways. Either the elements appear on separate lines (as shown above) or they ALL must appear on the same line (see example in SQ11). There are ways around this restriction, but they're not pretty.

You can string together IF statements. Suppose you want to read a file consisting of several protein sequences. The file looks like this:

```
>INSULIN - HUMAN (GI:4557671)
MALWMRLLPLLALLALWGPDPAAAFVNQHLCGSHLVEALYLVCGERGFFYTPKTRREAED
LQVGQVELGGGPGAGSLQPLALEGSLQKRGIVEQCCTSICSLYQLENYCN

>INSULIN - COW (GI:124564)
MALWTRLRPLLALLALWPPPPARAFVNQHLCGSHLVEALYLVCGERGFFYTPKARREVEG
PQVGALELAGGPGAGGLEGPPQKRGIVEQCCASVCSLYQLENYCN
```

```
>INSULIN - ZEBRA FISH (GI:18858895)
MAVWLQAGALLVLLVVSSVSTNPGTPQHLCGSHLVDALYLVCGPTGFFYNPKRDVEPLLG
FLPPKSAQETEVADFAFKDHAELIRKRGIVEQCCHKPCSIFELQNYCN
```

To teach a program how to interpret this file, you have to devise a strategy yourself. If you could see only one line at a time, how would you recognize the beginning of the protein? How would you recognize its end?

**SQ14: Pause for a moment and devise such a strategy.**

Here's a program based on one strategy:

```
IF line$[1:1] = ">" THEN
   LET header$ = line$
   LET beginning_of_protein = true
ELSE IF line$[1:1] = " " OR line$[1:1] = "" THEN
   LET end_of_protein = true
ELSE
   CALL Add_aminoacids_to_sequence(line$)
END IF
```

You can also nest IF statements (as shown in *Simulated_Site_search.Tru*).

**SQ15: The program below is supposed to determine whether a name is a nickname or a full name. Predict the output of the program for a variety of different names, then run the program to see if it works the way you think it should.**

```
DECLARE FUNCTION Right$

! ********* Input values ************
LET name$ = "Horace"
LET gender$ = "male"
LET syllables = 2

! ********* Main program ************
IF Right$(name$,1) = "y" THEN
   IF gender$ = "male" THEN
      PRINT "nickname"
   ELSE
      PRINT "full name"
   END IF
ELSE
   IF syllables = 1 THEN
      PRINT "nickname"
   ELSE
      PRINT "full name"
   END IF
END IF

FUNCTION Right$(string$) = string$[Len(string$):Len(string$)

END
```

## II.E. Loops

- Two kinds of loops: `FOR` and `DO`
- `FOR` *variable* `=` *begin* `TO` *end* `STEP` *i*
  *statements*
  `NEXT` *variable*
- The clause `STEP` *i* is optional
- `DO WHILE` *condition* `UNTIL` condition
  *statements*
  `LOOP WHILE` *condition* `UNTIL` condition
- The WHILE and UNTIL clauses are optional
- You can exit loops from within by `EXIT FOR` or `EXIT DO` statements

We haven't run across much in the short programs in these notes that a human couldn't do better than a computer. Computers come into their own when asked to perform operations repetitively, like millions of times. You can see this illustrated in the following loop you saw in the program *Simulated_Site_Search.Tru* (and reproduced below).

The loop specifies that the number 1 is initially written into the box called `position`. The subsequent statements are performed until `NEXT position` is encountered. Then `position` is incremented by 1 (2 is written in the box) and control loops back to the beginning, to the statement `LET base = ...` (hence the name "loop"). Once `position` has been incremented so many times that it equals `size_of_sequence`, control passes outside of this loop. Clearly, if the size of the sequence runs into the millions of nucleotides, as well it might, then this loop is something a human would not want to do by hand!

```
FOR position = 1 TO size_of_sequence
   LET base = Random_Integer(1,4)
   !   base = 1 for A, 2 for C, 3 for G, 4 for T
   LET sequence$[position:position] = bases$[base:base]
NEXT position
```

**SQ16. The following program prints the first 100 integers and their squares and square roots. Alter the program so that it prints the same information for the second 100 integers.**

```
FOR n = 1 TO 100
   PRINT n, n*n, Sqr(n)
NEXT n
END
```

`FOR ... NEXT` loops are convenient when statements are to be performed a set number of times. However, there are often cases where you don't know the number of desired iterations (compare "Drive 7 blocks" with."Drive until you see the red firehouse on the corner"). True Basic provides a second means of performing loops, as illustrated by the code below (inspired by code in *Simulated_Site_Search.Tru*):

```
LET first_base = 1
DO
   (search for a certain target within the sequence)
   (the value of first_base changes during the search)
LOOP UNTIL first_base > size_of_sequence
```

The same task might have been accomplished with an almost equivalent loop:

```
LET first_base = 1
DO WHILE first_base <= size_of_sequence
    (search for a certain target within the sequence)
    (the value of first_base changes during the search)
LOOP
```

(note that "<=" is not an arrow but means "less than or equal"). You're free to use either form (or many other methods), depending on what makes more sense to you.

**SQ17. Run the program listed below, then remove the "!" before DO and LOOP and change the program so that it doesn't crash when you enter a negative number**

```
! Displays square root of number provided by user
! Program ends when 0 is entered

DO
   ! DO
      INPUT PROMPT "Enter nonnegative number: ": n
   ! LOOP
   IF n = 0 THEN EXIT DO
   PRINT Sqr(n)     ! Sqr() is a built-in function for square-root
LOOP UNTIL n = 0

END
```

### II.F. Subroutines and functions

You've tied a lot of shoes in your lifetime. If you did so according to a single DO or FOR loop, it might look like:

```
FOR shoe = 1 TO total_shoes
    (Instructions how to tie shoe)
NEXT shoe
```

If so, you'd tie all your shoes all at one sitting and then get on with your life. While efficient, this is often not the best way to organize one's life; nor is it often the best way to organize one's programs. True Basic offers multiple ways of reusing instructions intermittently. I'll discuss two related tools: Subroutines and Functions.[2]

- Arguments to subroutines are passed by reference, unless the variables are enclosed in parentheses
- Arguments to functions are passed only by value
- Functions return only a single value
- If the name of a function ends in $, a string is returned, otherwise a number
- Functions must be declared or defined prior to invocation
- The scope of variables in the main program extend to internal subroutines and functions unless otherwise specified

***Subroutines*** are blocks of instructions that you can call upon from anywhere in the program. For this to work, the program has to be ***defined***, with a block of instructions between one set of SUB and END SUB statements, and has to be ***invoked***, with one or more CALL statements.

[2] True Basic also supports Pictures (for code invoked to draw graphic images with optional transformations) and Handlers (for code invoked when an error in executing the program occurs)

```
! *********** CONSTANTS ************
LET locale$ = "home"

! ********** MAIN PROGRAM **********
! Morning activities
...
CALL Tie_shoes(sneakers)
...

! Activities before going to funeral
...
CALL Tie_shoes(black_shoes)
...

! **** SUBROUTINES AND FUNCTIONS ****
SUB Tie_shoes(specific_shoes)
   IF locale$ = "home" THEN
      (Instructions how to tie shoes at home)
   ELSE
      (Generic instructions how to tie shoes away from home)
   END IF
END SUB
```

Note that the invocation of the subroutine (the `CALL` statement) may provide within parentheses a list of additional information (typically variables or constants) to pass to the subroutine. Each invocation may pass different information. The information is ***associated*** with variables used by the subroutine. In the present example, one invocation of `Tie_shoe` passes the variable sneakers, while another passes the variable `black_shoes`. The subroutine doesn't care. Either way, it refers to the variable internally as `specific_shoes`. Any change it may make to `specific_shoes` is reflected in the variable passed by the invoking program.

**SQ18. Predict what the output of the program below, then run the program to confirm your prediction.**

```
LET fake_mRNA$ = "GAAGCUCUU"
LET original_fake_mRNA$ = fake_mRNA$
CALL Tail_message(fake_mRNA$)
PRINT original_fake_mRNA$, fake_mRNA$

! **** SUBROUTINES AND FUNCTIONS ****
SUB Tail_message(mRNA$)
   LET length_of_mRNA = Len(mRNA$)
   LET mRNA$[length_of_mRNA:length_of_mRNA+10]= "AAAAAAAAAA"
END SUB

END
```

***Functions*** work similarly in many respects as subroutines. The main difference is that, like mathematical functions, True Basic functions return a value. For example:

```
! ****** LIBRARIES AND DECLARATIONS ******
DECLARE FUNCTION Tie_shoe

! ********** MAIN PROGRAM **********
LET tied_sneakers = Tie_shoe(sneakers)

! **** SUBROUTINES AND FUNCTIONS ****
FUNCTION Tie_shoes(specific_shoes)
   (Instructions how to tie shoes)
   LET Tie_shoes = tied_shoes
END FUNCTION
```

Functions are invoked not by CALL statements but merely by the appearance of their names. The name is replaced by a value specified by the function (which assigns a value to the name of the function), and that value may be used by the invoking program as any constant is used. Since there is no handy word like CALL to warn True_basic that a function is being invoked, functions need to be DECLAREd before use. Otherwise the compiler cannot distinguish between a function invocation and the appearance of a new variable. Unlike subroutines, functions can't change the variables they are passed.

**SQ19. Predict what the output of the program below, then run the program to confirm your prediction.**

```
DECLARE FUNCTION Tail_message$

LET fake_mRNA$ = "GAAGCUCUU"
LET original_fake_mRNA$ = fake_mRNA$
LET tailed_message$ = Tail_message(fake_mRNA$)
PRINT original_fake_mRNA$, fake_mRNA$, tailed_message$

! **** SUBROUTINES AND FUNCTIONS ****
FUNCTION Tail_message$(mRNA$)
   LET length_of_mRNA = Len(mRNA$)
   LET mRNA$[length_of_mRNA:length_of_mRNA+10]= "AAAAAAAAAA"
   LET Tail_message$ = mRNA$
END FUNCTION

END
```

As SQ18 and SQ19 illustrate, subroutines and functions are often interchangeable. You'll find later that subroutines are more powerful, but functions often aid in the readability of the program.

## III. Structured programming

### III.A. The need for structured programming

One advantage of personal programming is that you don't have to adhere to any rules or formats. After all, there's no other person who needs to understand the program.

Wrong.

No matter how crystalline the logic of your program may appear right now, in a surprisingly short period of time your clarity of vision will disappear into the sands of time and YOU will be that other person trying vainly to figure out how in the world your program works. You will want to fix errors in your program, make improvements, or describe to others how you made your marvelous discoveries. You will stare blankly at opaque code or struggle to piece together shards of logic like an archaeologist trying to comprehend an ancient civilization.

Don't let this happen to you! The time invested in writing organized, self-documenting programs repays itself many times over, and you should at the outset make this investment a habit.

A case in point. Examine and run the working program called *Mystery_program1.Tru*. Can you guess what it's trying to do? Could you readily make changes to the program if you wished to alter its function? Now look at the program called *Mystery_program2.Tru.* Believe it or not, the two programs are logically equivalent.

**SQ20. Alter *Mystery_program2.Tru* so that odd-length sequences are always determined to be nonpalindromic. (Note: you can do this even if you're not clear about what palindromes are or what of odd-length sequences are)**

### III.B. The process of writing a structured program

There are several important elements in writing programs that will enable you to comprehend them, maintain them, and improve them, even when you've become a stranger to them.

III.B.1. Descriptive names for variables, subroutines, and functions
What's this statement mean?

```
if a(ucase$(s$)) = z then ...
```

Who knows? But you have a chance with this logically equivalent statement:

```
IF palindrome_test(uppercase_sequence$) = true THEN ...
```

It does take more time to type descriptive names of variables, but you save pounds of brain cells later (and the time to regrow them) otherwise spent trying to figure out incomprehensible code. True Basic provides a tool to help you do the right thing. If you like, you can write the entire program with some variable named `x`, then once that's done you can issue the following command in the command box:

```
CHANGE x, length_of_sequence
```

Immediately, all instances of the variable `x` are changed to `length_of_sequence`.

III.B.2. Indentation and capitalization

Bring up the program called *Purine_tract.Tru*. Here's an excerpt from it:

```
do
if sequence$[left:left] = "A" or sequence$[left:left] = "G" then
let tract_found = false
for right = left+1 to length_of_sequence
if sequence$[right:right] <> "A" and  sequence$[right:right] <> "G" then
let tract_found = true
exit for
```

You have to admit that the variable names are very nice… and still this program certainly does not explode with meaning in your mind. Now go to the True Basic's command box and type:

```
DO Format
```

When you press enter, the program suddenly becomes much clearer! Mind you, it's still not a well structured program, but it's world's better than it was a moment ago.

The primary virtue of the indented program is not mere beauty but functionality. I confess, I tried to write the original unindented program as you saw it, but I couldn't. It was too difficult to trace down logical errors. So I wrote it properly formatted and then UNformatted it! You should never have to use True Basic's Format program (except on ***other*** people's programs). Your life will be so much easier if you indent as you write.

It is also useful to have capitalization conventions. Unlike many Unix-inspired languages, True Basic doesn't make distinctions as to upper and lower case. The following two statements work equally well, so far as the language is concerned:

```
IF base < end THEN LET base = Complement(base)
iF bAsE < eNd ThEn LeT bAsE = cOmPlEmEnT(bAsE)
```

I think, however, that readability is substantially improved by adopting the standard conventions of capitalizing all key words, leaving all variables in lower case, and capitalizing the first letters of functions and subroutines.

### III.B.3. Modularizing the program / Writing in small chunks

Grandmasters are good at chess not because they can try out mental moves so much faster than normal folk but because they recognize important patterns and think in chunks of moves. Expert typists can't think letters or move their fingers much better than the rest of us, but they can type whole words at once, in chunks. Ditto with pianists and chunks of notes. The fact is, we humans can't keep a whole lot in our heads all at once. We do best when we construct a hierarchy of concepts which we can manipulate at different levels.

Suppose I had the (rather ambitious) desire to write a computer simulation of my life. I could teach the computer how to represent every last thing I did, but if I tried this in a linear fashion, I wouldn't make it past 9:15 am without losing track of what I taught it about 8:30. A better approach is to set down the big picture – all the things I want to teach the computer. For example:

```
! Day_in_the_life
...
! ************** MAIN PROGRAM *************
CALL Wake_up
CALL Get_out_of_bed
CALL Drag_a_comb_across_my_head
...
! ******* SUBROUTINES and FUNCTIONS *******
SUB Wake_up
END SUB
SUB Get_out_of_bed
END SUB
SUB Drag_a_comb_across_my_head
END SUB
```

In writing my program this way, I accomplish several important things: (1) I can see at a glance of the Main Program all that the program intends to accomplish, (2) I have broken down a behemoth task into small, doable chunks, (3) I have thereby made the debugging process much easier, since I can test each component to my satisfaction before moving on to the next chunk. For this program to run without True Basic objecting, it's necessary to put in fake subroutine definitions, called ***stubs***. Initially, they can be empty or if it helps debugging, you can put within them messages like `PRINT "Wake_up called".`

The process of modularization can be extended to the program as a whole. I've found it useful to organize programs around a template, which you can see in *Template.Tru*. There's nothing sacred about this organizational scheme. If you have one that works better, by all means use it. But if you ***don't*** have one that works ***better***, you're highly advised to use this one. Time spent organizing code is time not spent rewriting forgotten code.

If you revisit the programs entitled *Mystery_program1.Tru* and *Mystery_program2.Tru,* you'll note that one is considerably longer than the other. It obviously took significantly greater effort to write the second program than the first. The thought of expending such effort for every program must give one pause.

The payoff is that code that is well organized, well documented, and well tested, is code that can be well used and reused. True Basic permits you to incorporate prewritten code into your program with LIBRARY statements. Need to test for palindromic sequences in a program you're writing? Already wrote such a routine? Then insert the following statement into your program:

```
LIBRARY "Palindrome_test.Tru"
```

Whatever is in that file is now available to your program. If you invest time in building good, modular tools, you can rapidly piece them together to make complicated but reliable programs.

### III.B.4. Documentation

You may have a program organized to the highest degree, with variables cleverly named and the most exquisite indentation, but still not be able to understand what the program's trying to do. There's sometimes no substitute for straightforward comments at critical junctures in the program. At minimum, the beginning of the program should contain a description of the overall purpose of the program, it's broad strategy, and (if applicable) the format of the input and output. It isn't overkill to provide the same information for each function and subroutine.

While documentation takes time, bear in mind that a quick-and-dirty program is generally weeks, maybe days, away from turning into an old-and-dead program.