

Programming the HP 49 G Calculator in User RPL Language

By

Gilberto E. Urroz, Ph.D., P.E.

Distributed by

 ***infoClearinghouse.com***

©2000 Gilberto E. Urroz
All Rights Reserved

PROGRAMMING THE HP 49 G	2
Examples of sequential programming	2
Programs generated by defining a function.	2
Programs that mimic a sequence of calculator operations.	4
Interactive input in programs	6
Prompt with an input string	7
A function with an input string	8
Debugging the program	9
Fixing the program	9
Input string for programs requiring two or three input values	10
Input string program for two input values	11
Input string program for three input values	13
Identifying output in programs	14
Tagging a numerical result	14
Decomposing a tagged numerical result into a number and a tag	15
“De-tagging” a tagged quantity	15
Using a message box for output	17
Including input and output in a message box – string concatenation	18
Incorporating units within a program	19
Relational and logical operators	22
Relational operators	22
Logical operators	23
Program branching	24
What IF...?	24
IF...THEN...END	24
IF...THEN...ELSE...END	26
Nested IF...THEN...ELSE...END constructs	27
Just in CASE ...	28
Program loops	30
START	30
START...NEXT	30
START...STEP	33
FOR	33
FOR...NEXT	34
FOR...STEP	34
DO	35
WHILE	36
Procedural programming and object-oriented programming	37
Concluding remarks on programming	37
REFERENCES (for all HP49 documents at InfoClearinghouse.com)	38

Programming the HP 49 G

In this document we present the basics of programming the HP 49 G in User RPL language through a variety of examples. I recommend that you try the programming examples presented in **“Functions, Lists, and Programs with the HP 49 G Calculator”** before reading this document. It is not the intention of this chapter to teach everything about programming the calculator. It is instead aimed to providing programming tools and examples that can be used later in the development of simple programming for evaluating functions, data processing, and even graphs.

Examples of sequential programming

Some very simple programming examples in *User RPL* language were presented in **“Functions, Lists, and Programs with the HP 49 G Calculator”**. In general, a program is any sequence of calculator instructions enclosed between the program symbols << and >>. The examples presented in the document mentioned above can be classified basically into two types:

Programs generated by defining a function.

These are programs generated by entering a function definition of the form:

```
'function_name(x1, x2, ...) = expression containing variables x1, x2, ...)'
```

and using the keystroke sequence

[←][DEF].

The program is stored in a variable called `function_name`. When the program is recalled to the stack, by using

[→][`function_name`]

the program shows up as follows:

```
<< → x1, x2, ... 'expression containing variables x1, x2, ...'>>.
```

To evaluate the function for a set of input variables x_1, x_2, \dots , enter the variables into the stack in the appropriate order (i.e., x_1 first, followed by x_2 , then x_2 , etc.), and press the soft key labeled [function_name]. The calculator will return the value of the function $\text{function_name}(x_1, x_2, \dots)$.

Example: *Manning's equation for wide rectangular channel.*

As an example, consider the following equation that calculates the unit discharge (discharge per unit width), q , in a wide rectangular open channel using Manning's equation:

$$q = \frac{C_u}{n} y_0^{5/3} \sqrt{S_0},$$

where C_u is a constant that depends on the system of units used [$C_u = 1.0$ for units of the International System (S.I.), and $C_u = 1.486$ for units of the English System (E.S.)], n is the Manning's resistance

coefficient, which depends on the type of channel lining and other factors, y_0 is the flow depth, and S_0 is the channel bed slope given as a dimensionless fraction.

Note: Values of the Manning's coefficient, n , are available in tables as dimensionless numbers, typically between 0.001 to 0.5. The value of C_u is also used without dimensions. However, care should be taken to ensure that the value of y_0 has the proper units, i.e., m in S.I. and ft in E.S. The result for q is returned in the proper units of the corresponding system in use, i.e., m^2/s in S.I. and ft^2/s in E.S. Manning's equation is, therefore, not *dimensionally consistent*.

Suppose that we want to create a function $q(C_u, n, y_0, S_0)$ to calculate the unit discharge q for this case. We will proceed as follows:

[EQW] [ALPHA][←][Q] [←][∘] [ALPHA][C] [ALPHA][←][U] [SPC] [ALPHA][←][N] [SPC]
[ALPHA][←][Y][0] [SPC] [ALPHA][S][0] [▲][▲] [↵][=] [ALPHA][C] [ALPHA][←][U] [÷]
[ALPHA][←][N] [▶] [×][ALPHA][←][Y][0] [y^x] [5] [÷] [3] [▶][▶] [×] [√x] [ALPHA][S][0]
[ENTER]

The expression entered may look like this:

$$q(C_u, n, y_0, S_0) = C_u / n * y_0^{(5/3)} * \sqrt{S_0},$$

if you have not selected the textbook display option. Otherwise, it will look like the equation shown earlier.

To define the function use:

[←][DEF]

Press [VAR], if needed, to recover the variable list. At this point there will be a variable called [q] in your soft key menu. To see the contents of q , use

[↵][q].

The program generated by defining the function $q(C_u, n, y_0, S_0)$ shows up as:

<< → C_u n y_0 S_0 ' $C_u/n*y_0^{(5/3)}*\sqrt{S_0}$ ' >>.

This is to be interpreted as

“enter C_u , n , y_0 , S_0 , in that order, then calculate the expression between quotes.”

For example, to calculate q for $C_u = 1.0$, $n = 0.012$, $y_0 = 2$ m, and $S_0 = 0.0001$, use:

[1][ENTER] [.] [0][1][2][ENTER] [2][ENTER] [.] [0][0][0][1][ENTER] [q]

The result is 2.6456684 (or, $q = 2.6456684 \text{ m}^2/\text{s}$).

You can also separate the input data with spaces in a single stack line rather than using [ENTER]. For example, to calculate q given $C_u = 1.486$, $n = 0.022$, $y_0 = 3.2$ m, and $S_0 = 1 \times 10^{-3}$, use:

[1][.] [4][8][6] [SPC] [.] [0][2][2] [SPC] [3][.] [2] [SPC] [1][EEX][3][+/-] [q]

The result now is 14.8426948578 (or, $q = 14.8426948578 \text{ ft}^2/\text{s}$).

Programs that mimic a sequence of calculator operations.

In this case, the terms to be involved in the sequence of operations are assumed to be present in the stack. The program is typed in by first opening the program symbols with [↵][<>]. Next, the sequence of operations to be performed is entered. When all the operations have been typed in, press [ENTER] to complete the program. If this is to be a once-only program, you can at this point, press [↵][EVAL] to execute the program using the input data available. If it is to be a permanent program, it needs to be stored in a variable name.

The best way to describe this type of programs is with an example:

Example: Velocity head for a rectangular channel.

Suppose that we want to calculate the velocity head, h_v , in a rectangular channel of width b , with a flow depth y , that carries a discharge Q . The specific energy is calculated as

$$h_v = \frac{Q^2}{2g(by)^2},$$

Where g is the acceleration of gravity ($g = 9.806 \text{ m/s}^2$ in S.I. units or $g = 32.2 \text{ ft/s}^2$ in E.S. units). If we were to calculate h_v for $Q = 23 \text{ cfs}$ (cubic feet per second = ft^3/s), $b = 3 \text{ ft}$, and $y = 2 \text{ ft}$, we would use:

$$h_v = \frac{23^2}{2 \cdot 32.2 \cdot (3 \cdot 2)^2}.$$

Using the RPN system in the HP 49 G, interactively, we can calculate this quantity as:

[2] [ENTER] [3] [×] [↵][x^2] [3][2][.] [2] [×] [2] [×] [2][3][↵][x^2] [►] [÷]

Resulting in 0.228174, or $h_v = 0.228174 \dots \text{ ft}$. (Since we didn't use units in the input, the result will have no units either).

Notice that we start by calculating the last term in the denominator, then entering the next term to the left. Next, we multiply those two terms. Next we enter 2 and multiply it with register x . Finally, we enter the numerator, swap the order, and divide the two terms. Although this is not the only way to obtain this result, this approach is the most efficient in order to translate the procedure into a program because it uses mainly register x in the calculations.

To put this calculation together as a program we need to have the input data (Q , g , b , y) in the stack in the order in which they will be used in the calculation. In terms of the variables Q , g , b , and y , the calculation just performed is written as (do not type the following):

y [ENTER] b [×] [↵][x^2] g [×] [2] [×] Q [↵][x^2] [►] [÷]

As you can see, y is used first, then we use b , g , and Q , in that order. Therefore, for the purpose of this calculation we need to enter the variables in the inverse order, i.e., (do not type the following):

Q [ENTER] g [ENTER] b [ENTER] y [ENTER]

For the specific values under consideration use:

[2][3][ENTER] [3][2][.] [2][ENTER] [3][ENTER] [2][ENTER]

The program itself will contain only those keystrokes (or commands) that result from removing the input values from the interactive calculation shown earlier, i.e., removing Q, g, b, and y from (do not type the following):

y [ENTER] b [×] [↵][x²] g [×] [2] [×] Q [↵][x²] [▶] [÷]

and keeping only the operations shown below (do not type the following):

[×] [↵][x²] [×] [2] [×] Q [↵][x²] [▶] [÷]

Note: When entering the program do not use the keystroke [▶], instead use the keystroke sequence: [↵] [PRG][STACK][SWAP].

However, unlike the interactive use of the calculator performed earlier, we need to do some swapping of registers x and y within the program. To write the program, we use, therefore:

[↵][<>]	Opens program symbols
[×]	Multiply y (register x) with b (register y)
[↵][x ²]	Square (b·y)
[×]	Multiply (b·y) ² [register x] times g [register y]
[2] [×]	Enter a 2 [now in reg x] and multiply it with g·(b·y) ² [now in reg. y]
[↵][PRG][STACK][SWAP]	Swap Q [reg. y] with 2·g·(b·y) ² [now in reg. y]
[↵][x ²]	Square Q
[↵][PRG][STACK][SWAP]	Swap 2·g·(b·y) ² [reg. y] with Q ² [reg. x]
[÷]	Divide Q ² [reg. y] by 2·g·(b·y) ² [reg. x]
[ENTER]	Enter the program

The resulting program looks like this:

<< * SQ * 2 * SWAP SQ SWAP / >>

Note: SQ is the function that results from the keystroke sequence [↵][x²].

Let's make an extra copy of the program and save it into a variable called hv:

[ENTER] [↵]['] [ALPHA][↵][H] [ALPHA][↵][V] [STO▶]

A new variable [hv] should be available in your soft key menu. (Press [VAR] to see your variable list.) Also, check that your stack looks like this:

5:	23.
4:	32.2
3:	3.
2:	2.
1:	<< * SQ * 2 * SWAP SQ SWAP / >>

To evaluate the program use [↵][EVAL]. The result should be 0.228174..., as before.

The program is available for future use in variable [hv]. For example, for Q = 0.5 m³/s, g = 9.806 m/s², b = 1.5 m, and y = 0.5 m, use:

[.][5][SPC] [9][.][8][0][6][SPC] [1][.][5][SPC] [.][5] [hv]

Note: [SPC] is used here as an alternative to [ENTER] for input data entry.
--

The result now is 2.26618623518E-2, i.e., $h\nu = 2.26618623518 \times 10^{-2}$ m.

Since the equation programmed in [hv] is dimensionally consistent, we can use units in the input. Try the following:

```
[.] [5] [→] [UNITS] [VOL][m^3] [1][→] [UNITS] [TIME][ s ] [+]  
[9][.] [8][0][6] [→] [UNITS][LENG][ m ] [1][→] [UNITS] [TIME][ s ] [←][x^2] [+]  
[1][.] [5] [→] [UNITS][LENG][ m ]  
[.] [5] [→] [UNITS][LENG][ m ]  
[VAR][ hv ]
```

The result, now with units attached, is 2.26618623518E-2_m.

The two types of programs presented in this section are *sequential programs*, in the sense that the program flow follows a single path, i.e., INPUT → OPERATION → OUTPUT. Branching of the program flow is possible by using the commands in the menu [←][PRG][BRCH], i.e., IF, CASE, START, FOR, DO, WHILE, and the functions IFT and IFTE (examples of the latter were also presented in chapter 6). More detail on program branching is presented below.

Interactive input in programs

In the sequential program examples shown in the previous section it is not always clear to the user the order in which the variables must be placed in the stack before program execution. For the case of the program *q*, written as

```
<< → Cu n y0 S0 'Cu/n*y0^(5/3)*√S0' >>,
```

it is always possible to recall the program definition into the stack ([→][q]) to see the order in which the variables must be entered, namely,

```
→ Cu n y0 S0.
```

However, for the case of the program [hv], its definition

```
<< * SQ * 2 * SWAP SQ SWAP / >>,
```

does not provide a clue of the order in which the data must be entered, unless, of course, you are extremely experienced with RPN and the User RPL language.

One way to check the result of the program as a formula is to enter symbolic variables, instead of numeric results, in the stack, and let the program operate on those variables. For this approach to be effective the calculator's CAS (Calculator Algebraic System) must be set to symbolic and exact modes. This is accomplished by typing

```
[MODE][CAS],
```

and ensuring that the check marks in the options

```
_Numeric and _Approx
```

are removed. Press [OK][OK] to return to normal calculator display. Press [VAR] to display your variables menu.

We will use this latter approach to check what formula results from using the program [hv] as follows: We know that there are four inputs to the program, thus, we use the symbolic variables S4, S3, S2, and S1 to indicate the stack levels at input:

[ALPHA][S][4][ENTER] [ALPHA][S][3][ENTER] [ALPHA][S][2][ENTER] [ALPHA][S][1][ENTER]

Next, press [hv]. The resulting formula may look like this

$$\text{'SQ(S4) / (S3*SQ(S2*S1)*2) '},$$

if your display is not set to textbook style, or like this,

$$\frac{SQ(S4)}{S3 \cdot SQ(S2 \cdot S1) \cdot 2},$$

if textbook style is selected. Since we know that the function SQ() stands for x^2 , we interpret the latter result as

$$\frac{S4^2}{2 \cdot S3 \cdot (S2 \cdot S1)^2},$$

which indicates the position of the different stack input levels in the formula. By comparing this result with the original formula that we programmed, i.e.,

$$h_v = \frac{Q^2}{2g(by)^2},$$

we find that we must enter y in stack level 1 (S1), b in stack level 2 (S2), g in stack level 3 (S3), and Q in stack level 4 (S4).

Prompt with an input string

These two approaches for identifying the order of the input data are not very efficient. You can, however, help the user identify the variables to be used by prompting him or her with the name of the variables. From the various methods provided by the User RPL language, the simplest is to use an input string and the function INPUT ([↵][PRG][NXT][IN][INPUT]) to load your input data.

The following program prompts the user for the value of a variable a and places the input in stack level 1:

```
<< "Enter a: " {↵:a: " {2 0} V } INPUT OBJ→ >>
```

To type this program use the following:

[↵][<>][↵][“”]	Open program symbol, open double quotes (string).
[ALPHA][↵][ALPHA]	Lock alpha keyboard in lower case
[ALPHA][ALPHA]	Lock keyboard in alphabetic mode
[↵][E][N][T][E][R][SPC][A][↵][::]	Type in Enter a: "
[ALPHA]	Unlock alphabetic keyboard
[▶][↵][{}][↵][“”]	Type in { "

[↵][←][↵][::][ALPHA][A][▶][▶]	Type in ←:a: " (*)
[↵][{}][2][SPC][0][▶]	Type in {2 0}
[ALPHA][↵][V][▶]	Type in v }
[↵][PRG][NXT][IN][INPUT]	Enter INPUT
[NXT][PRG][TYPE][OBJ→]	Enter OBJ→
[ENTER]	Enter program in stack level 1

(*) The symbol ← indicates a line feed. This symbol is obtained by using [↵] with the [.] key.

Save the program in a variable called INPTa (for INPuT a) as follows:

[↵]['] [ALPHA][ALPHA] [I][N][P][T][↵] [A] [ENTER] [STO▶]

Press [VAR]. The variable [INPTa] should be available in your soft key menu.

Try running the program by pressing the soft key labeled [INTPa]. The result is a stack prompting the user for the value of a and placing the cursor right in front of the prompt :a: Enter a value for a, say 35, then press [ENTER]. The result is the input string

:a: 35

in stack level 1.

A function with an input string

If you were to use this piece of code to calculate the function, $f(a) = 2*a^2+3$, you could modify the program to read as follows:

<< "Enter a: " {←:a: " {2 0} v } INPUT OBJ→ → a << '2*a^2+3'>> >>

To modify the program use:

[↵][INTPa][▼]	Copies contents of INTPa to register x and launches editor
[↵][▼][←]	Sends cursor to end of program
[↵][→][ALPHA][↵][A]	Type in → a
[↵][<>][↵][']	Creates sub-program symbol and starts algebraic symbol (')
[2][×][ALPHA][↵][A][y^x][2][+][3]	Type in algebraic expression $2*a^2+3$
[ENTER]	

Save this new program under the name 'FUNCa' (FUNCtion of a):

[↵]['] [ALPHA][ALPHA] [F][U][N][C][↵] [A] [ENTER] [STO▶]

Run the program by pressing [FUNCa]. When prompted to enter the value of a enter, for example, 2, and press [ENTER]. The result is simply the algebraic $2a^2+3$, which is an incorrect result. The HP 49 G provides functions for debugging programs to identify logical errors during program execution as shown below.

Debugging the program

To figure out why it did not work we use the DEBUG program in the HP 49 G as follows:

[↵]['] [FUNCa][ENTER]	Copies program name to stack level 1
[↵][PRG][NXT][NXT][RUN][DEBUG]	Starts debugger
[SST↓]	Step-by-step debugging, result: "Enter a:"
[SST↓]	Result: { "↵ a:" {2 0} V }
[SST↓]	Result: user is prompted to enter value of a
[2][ENTER]	Enter a value of 2 for a. Result: "↵a:2"
[SST↓]	Result: a:2
[SST↓]	Result: empty stack, executing →a
[SST↓]	Result: empty stack, entering subprogram <<
[SST↓]	Result: '2*a^2+3'
[SST↓]	Result: '2*a^2+3', leaving subprogram >>
[SST↓]	Result: '2*a^2+3', leaving main program>>

Further pressing [SST↓] produces no more output since we have gone through the entire program, step by step. This run through the debugger did not provide any information on why the program is not calculating the value of $2a^2+3$ for $a = 2$. To see what is the value of a in the sub-program, we need to run the debugger again and evaluate a within the sub-program. Try the following:

[VAR]	Recovers variables menu
[↵]['] [FUNCa][ENTER]	Copies program name to stack level 1
[↵][PRG][NXT][NXT][RUN][DEBUG]	Starts debugger
[SST↓]	Step-by-step debugging, result: "Enter a:"
[SST↓]	Result: { "↵ a:" {2 0} V }
[SST↓]	Result: user is prompted to enter value of a
[2][ENTER]	Enter a value of 2 for a. Result: "↵a:2"
[SST↓]	Result: a:2
[SST↓]	Result: empty stack, executing →a
[SST↓]	Result: empty stack, entering subprogram <<

At this point we are within the subprogram << '2*a^2+3'>> which uses the local variable a. To see the value of a use:

[ALPHA][↵][A] [↵][EVAL]	This indeed shows that the local variable a = 2
-------------------------	---

Let's kill the debugger at this point since we already know the result we will get. To kill the debugger press [KILL]. You receive an <!> Interrupted message acknowledging killing the debugger. Press [ON] to recover normal calculator display.

Note: In debugging mode, every time we press [SST↓] the top left corner of the display shows the program step being executed. A soft key function called [SST] is also available under the [RUN] menu. This can be used to execute at once any sub-program called from within a main program. Examples of the application of [SST] will be shown later.

Fixing the program

Let's list the program in the stack once more:

[VAR]	Recovers variables menu
[↵][FUNCa]	Copies program to stack level 1

The only possible explanation for the failure of the program to produce a numerical result seems to be the lack of an EVAL function after the algebraic expression '2*a^2+3'. Let's edit the program by adding the missing EVAL function as follows:

[▼]	Launch editor
[↵][▼][◀][◀][◀]	Sends cursor to end of algebraic expression
[↵][EVAL]	Evaluate algebraic
[ENTER]	Enter edited program back into stack level 1
[↵][FUNCa]	Shortcut to store level 1 into variable FUNCa

Run the program again by pressing [FUNCa]. When prompted for the value of a enter 2, and press [ENTER]. The result is now 11, the correct result.

The modified function program will look like this (use [↵][FUNCa] to recall its contents to the stack):

```
<< "Enter a: " { "↵":a: " {2 0} V } INPUT OBJ→ → a << '2*a^2+3' EVAL >> >>
```








Input string for programs requiring two or three input values

In this section we will create a sub-directory, within the directory HOME, to hold examples of input strings for one, two, and three input data values. These will be generic input strings that can be incorporated in any future program, taking care of changing the variable names according to the needs of each program.

Let's get started by creating a sub-directory called PTRICKS (Programming TRICKS) to hold programming tidbits that we can later borrow from to use in more complex programming exercises. To create the sub-directory, first make sure that you move to the HOME directory, by using [UPDIR] as many times as needed. Within the HOME directory, use the following keystrokes to create the sub-directory PTRICKS:

[↵]['][ALPHA][ALPHA][P][T][R][I][C][K][S] [ENTER]	Enter directory name PTRICKS
[↵][PRG][MEM][DIR][CRDIR]	Create directory
[VAR]	Recover variables listing in soft keys

Before we develop the programs for 2 and 3 variables, let's copy the program INPTa into PTRICKS as follows:

-  Move to the directory where you defined INPTa (skip this step if INTPa is defined in the HOME directory)
-  Copy contents of INPTa into stack level 1 by using [↵][INPTa].
-  Copy the name of the program to stack level 1 by using [↵]['][INPTa][ENTER]
-  Edit the text in level 1 to read INPT1 by using [▼][↵][▶][◀][↵][1][ENTER]
-  Move within directory PTRICKS by pressing [PTRICKS] in the HOME directory (move back to the HOME directory before performing this step if needed)
-  You should have the program in stack level 2 and the program name, INPT1, in stack level 1. Press [STO▶] to store the program into the appropriate variable name.
-  Press [VAR] to recover the list of variables within PTRICKS. There should be only one variable listed, namely, INPT1.

Note: A procedure similar to this can be used to easily copy a variable from one sub-directory to another. You can even copy the contents of a full directory to stack level 1 by using [↵] followed by the soft key corresponding to the directory name. For example, moving back to the HOME directory you could copy the contents of PTRICKS onto stack level 1 by using [↵][PTRICKS]. At this moment, you should get the following in stack level 1:

```
DIR
INPT1 << "Enter a: "
{"↵:a: " {2 0} V
} INPUT OBJ→ >>
END
```

This is the format in which a directory is listed in the stack. The listing starts with the word DIR to identify the contents of the stack as a directory, followed by the names and contents of all the variables available in the directory, and finishing with the word END.

A program may have more than 3 input data values, however, when using input strings we want to limit the number of input data values to 3 at a time for the simple reason that, in general, we have visible only 5 stack levels. If we use stack level 5 to give a title to the input string, and leave stack level 4 empty to facilitate reading the display, we have only stack levels 1 through 3 to define input variables. [Note: With the use of a smaller font we can get more than 5 levels in the stack, thus allowing form more than 3 variables to be input at a time, but with the standard HP 49 G display format the default is 5 stack levels.] If we have more than 3 variables to input, we can use a second, third, etc., input string to enter the remaining input values. Having available programs for entering 1, 2, and 3 input values per input string we can handle most inputs to our programs.

Input string program for two input values

The input string program for two input values, say a and b, looks as follows:

```
<< "Enter a and b: " {"↵:a:↵:b: " {2 0} V } INPUT OBJ→ >>
```

This program can be easily created by modifying the contents of INPT1 as follows:

[↵][INPT1]	Copy contents of INPT1 to stack level 1
[▼]	Start editor
[▶](10 times)	Move cursor to the right of the letter a
[ALPHA][↵][ALPHA]	Lock alpha keyboard in lower case
[ALPHA][ALPHA]	Lock keyboard in alphabetic mode
[SPC][A][N][D][SPC][B]	Type in and b
[ALPHA]	Unlock alphabetic keyboard
[▶](10 times)	Move cursor to the right of the character :
[↵][↵][↵][:][ALPHA][B]	Type in ↵ :b:"
[ENTER]	Enter program in stack
[↵]['][INPT1]	Enter variable name INPT1 in stack
[◀][↵][2][ENTER]	Edit variable name to read INPT2
[STO▶]	Store new program in variable INPT2
[VAR]	To recover variables list






To check the operation of the new program, press [INPT2]. Enter a value for a, say 2, (careful here), then press [▼], not [ENTER], to move the cursor in front of the b prompt, and enter the value of b, say 3. Now, press [ENTER]. The result is the tagged values a: 2 and b: 3 in stack levels 2 and 1, respectively.

Application: evaluating a function of two variables

The ideal gas law was introduced in Chapter 3:

$$pV = nRT,$$

where:

-  p = gas pressure (Pa),
-  V = gas volume(m³),
-  n = number of moles (gmol),
-  R = universal gas constant = 8.31451_J/(gmol*K), and
-  T = absolute temperature (K).

We can define the pressure p as a function of two variables, V and T, as $p(V,T) = nRT/V$ for a given mass of gas since n will also remain constant. Assume that n = 0.2 gmol, then the function to program is

$$p(V,T) = 8.31451 \cdot 0.2 \cdot \frac{T}{V} = (1.662902 - \frac{J}{K}) \cdot \frac{T}{V}.$$

We can define the function by typing the function definition as:

```
[EQW][ALPHA][←][P][←][()][ALPHA][V][SPC][ALPHA][T] [▲][▲] [→][=][←][()][
[1][.][6][6][2][9][0][2][→][_][ALPHA][J][÷][ALPHA][K][▶][▶][▶]
[×] [ALPHA][T][+][ALPHA][V][ENTER]
```

The function is defined by using [←][DEF] which creates the variable [p]. The contents of this variable, which can be recalled by using [VAR][→][P], are shown as

```
<< → V T '1.662902*(J/K)*(T/V)' >>
```

Note: the keystroke sequence [→][_] did not have an effect in the equation writer, instead, a * (times) sign was included. This, however, does not produce the desired effect of defining J/K as units. Edit the program to read:

```
<< → V T '(1.662902_J/K)*(T/V)' >>
```

and store it back into variable [p]. The next step is to add the input string that will prompt the user for the values of V and T. To create this input stream we place the contents of variable INPT2 into the stack: [→][INPT2], producing:

```
<< "Enter a and b: " { "←:a:←:b:" {2 0} V } INPUT OBJ→ >>
```

Modify it to read:

```
<< "Enter V and T: " { "←:V:←:T:" {2 0} V } INPUT OBJ→ >>
```

Press [ENTER] to create an additional copy of the program, and press [→][EVAL] to run this modified version of INPT2. Enter values of V = 0.01_m^3 and T = 300_K. Before pressing [ENTER], the stack will look like this:

```

Enter V and T:

:V:0.01_m^3
:T:300_K

```

Press [ENTER] to get the result

```

3: <<Enter V and T: "...
2:          V:0.01_m^3
1:          T:300_K

```

This shows that the INPT2 program places the values of V and T in stack levels 2 and 1, respectively. This is the correct order for data input into the program [p], i.e., $\text{in} \rightarrow V \ T$. To drop the contents of level 1 and 2, press the backspace key, [\leftarrow], twice.

The next step is to copy the corresponding piece of code from the program listed in the stack to the function definition. This can be accomplished by using the following:

[∇]	Start the program editor
[\blacktriangleright][\leftarrow][BEGIN] (press the APPS key)	Position cursor at beginning of code to copy
[\leftarrow][∇][\blacktriangleleft]	Position cursor at end of code to copy
[\leftarrow][END] (press the MODE key)	Highlight code to copy
[\leftarrow][COPY] (press the VAR key)	Copy highlighted code
[ENTER]	End editing of this program
[\leftarrow][p][∇][\blacktriangleright]	Copy contents of p, start the editor, position cursor
[\leftarrow][PASTE] (in the NXT key)	Paste code copied earlier
[ENTER]	Enter new program in stack

The combined program will look like this:

```

<< "Enter V and T: " { $\leftarrow$  :V: $\leftarrow$  :T: " {2 0} V } INPUT OBJ $\rightarrow$   $\rightarrow$  V T
'(1.662902_J/K)*(T/V)' >>

```

Store the new program into variable p by using [\leftarrow][p].

To test the program, press [p]. Enter values of $V = 0.01_m^3$ and $T = 300_K$ in the input string, then press [ENTER]. The result is 49887.06_J/m^3. The units of J/m^3 were proven earlier to be equivalent to Pascals (Pa), the preferred pressure unit in the S.I. system.

Note: because we deliberately included units in the function definition, the input values must have units attach to them in input to produce the proper result.

Input string program for three input values

The input string program for three input values, say a ,b, and c, looks as follows:

```

<< "Enter a, b and c: " { $\leftarrow$  :a: $\leftarrow$  :b: $\leftarrow$  :c: " {2 0} V } INPUT OBJ $\rightarrow$  >>

```

This program can be easily created by modifying the contents of INPT2 to make it look like shown immediately above. The resulting program can then be stored in a variable called INPT3. With this program we complete the collection of input string programs that will allow us to enter one, two, or three data values. Keep these programs as a reference and copy and modify them to fulfill the requirements of new programs you write.

Application: evaluating a function of three variables

Suppose that we want to program the ideal gas law including the number of moles, n, as an additional variable, i.e., we want to define the function

$$p(V, T, n) = (8.31451 \frac{J}{K}) \frac{n \cdot T}{V},$$

and modify it to include the three-variable input string. The procedure to put together this function is very similar to that used earlier in defining the function p(V,T). The resulting program will look like this:

```
<< "Enter V, T, and n:" {" ␣ :V:␣ :T:␣ :n:" {2 0} V } INPUT OBJ→ →V T n
'(8.34451_J/(K*mol))*(n*T/V)' >>
```

Store this result back into the variable [p]. To run the program, press [p].

Enter values of V = 0.01_m^3, T = 300_K, and n = 0.8_mol. Before pressing [ENTER], the stack will look like this:

Enter V, T and n: :V:0.01_m^3 :T:300_K :n:0.8_mol
--

Press [ENTER] to get the result 200268.24_J/m^3, or 200268.24_Pa = 200.27 kPa.

Identifying output in programs

The simplest way to identify numerical program output is to “tag” the program results. A tag in the HP 49 G calculator is simply a string attached to a number. The string will be the name associated with the numeric result. For example, earlier on, when checking the operation of the input string programs INTPa (or INPT1) and INPT2, we obtained as results tagged numerical output such as :a:35.

Tagging a numerical result

To tag a numerical result you need to place the number in stack level 2 and the tagging string in stack level 1, then use the →TAG function ([↵][PRG][TYPE][→TAG]). For example, to produce the tagged result B:5., use:

[5][ENTER]	Enter the numerical result
[↵]["][ALPHA][B]	Enter the tagging string
[↵][PRG][TYPE][→TAG]	Tag numerical result

Decomposing a tagged numerical result into a number and a tag

To decompose a tagged result into its numerical value and its tag, simply use the \rightarrow OBJ function, available at \leftarrow [PRG][TYPE][\rightarrow OBJ]. The result of decomposing a tagged number with \rightarrow OBJ is to place the numerical value in stack level 2 and the tag in stack level 1. If you are interested in using the numerical value only, then you will drop the tag by using the backspace key \leftarrow .

“De-tagging” a tagged quantity

“De-tagging” means to extract the numerical quantity out of a tagged quantity. This function is accessed through the keystroke combination: \leftarrow [PRG][TYPE][NXT][DTAG]. For example, given the tagged quantity $a:2$, DTAG returns the numerical value 2.

Examples: tagging output from function

Example 1 – tagging output from function FUNCa

Let’s modify the function FUNCa, defined earlier, to produce a tagged output. Use \rightarrow [FUNCa] to recall the contents of FUNCa to the stack. The original function program reads

```
<< "Enter a: " { $\leftarrow$ :a: " {2 0} V } INPUT OBJ $\rightarrow$   $\rightarrow$  a << '2*a^2+3' EVAL >> >>
```

Modify it to read:

```
<< "Enter a: " { $\leftarrow$ :a: " {2 0} V } INPUT OBJ $\rightarrow$   $\rightarrow$  a << '2*a^2+3' EVAL "F"  $\rightarrow$ TAG >> >>
```

Store the program back into FUNCa by using \leftarrow [FUNCa]. Next, run the program by pressing [FUNCa]. Enter a value of 2 when prompted, and press [ENTER]. The result is now the tagged result $F:11$.

Example 2 – tagging input and output from function FUNCa

In this example we modify the program FUNCa so that the output includes not only the evaluated function, but also a copy of the input with a tag.

Use \rightarrow [FUNCa] to recall the contents of FUNCa to the stack:

```
<< "Enter a: " { $\leftarrow$ :a: " {2 0} V } INPUT OBJ $\rightarrow$   $\rightarrow$  a << '2*a^2+3' EVAL >> >>
```

Modify it to read:

```
<< "Enter a: " { $\leftarrow$ :a: " {2 0} V } INPUT OBJ $\rightarrow$   $\rightarrow$  a << '2*a^2+3' EVAL "F"  $\rightarrow$ TAG a  
SWAP>> >>
```

(Recall that the function SWAP is available by using \leftarrow [PRG][STACK][SWAP].) Store the program back into FUNCa by using \leftarrow [FUNCa]. Next, run the program by pressing [FUNCa]. Enter a value of 2 when prompted, and press [ENTER]. The result is now two tagged numbers $a:2$. in stack level 2, and $F:11$. in stack level 1.

Note: Because we use an input string to get the input data value, the local variable a actually stores a tagged value (:a:2, in the example above). Therefore, we do not need to tag it in the output. All what we need to do is place an a before the SWAP function in the subprogram above, and the tagged input is placed in the stack. It should be pointed out that, in performing the calculation of the function, the tag of the tagged input a is dropped automatically, and only its numerical value is used in the calculation.

To see the operation of the function FUNCa, step by step, you could use the DBUG function as follows:

[↵]['] [FUNCa][ENTER]	Copies program name to stack level 1
[↵][PRG][NXT][NXT][RUN][DEBUG]	Starts debugger
[SST↓]	Step-by-step debugging, result: "Enter a:"
[SST↓]	Result: { "↵a:" {2 0} V }
[SST↓]	Result: user is prompted to enter value of a
[2][ENTER]	Enter a value of 2 for a. Result: "↵a:2"
[SST↓]	Result: a:2
[SST↓]	Result: empty stack, executing →a
[SST↓]	Result: empty stack, entering subprogram <<
[SST↓]	Result: '2*a^2+3'
[SST↓]	Result: 11.,
[SST↓]	Result: "F"
[SST↓]	Result: F: 11.
[SST↓]	Result: a:2.
[SST↓]	Result: swap levels 1 and 2
[SST↓]	leaving subprogram >>
[SST↓]	leaving main program>>

Example 3 – tagging input and output from function p(V,T)

In this example we modify the program [p] so that the output tagged input values and tagged result.

Use [↵][p] to recall the contents of the program to the stack:

```
<< "Enter V, T, and n:" { "↵ :V:↵ :T:↵ :n:" {2 0} V } INPUT OBJ→ →V T n
      '(8.34451_J/(K*mol))*(n*T/V)' >>
```

Modify it to read:

```
<< "Enter V, T and n: " { "↵ :V:↵ :T: " {2 0} V } INPUT OBJ→ →V T n <<V T n
      '(8.34451_J/(K*mol))*(n*T/V)' EVAL "p" →TAG >> >>
```

Note: Notice that we have placed the calculation and tagging of the function p(V,T,n), preceded by a recall of the input variables V T n, into a sub-program [the sequence of instructions contained within the inner set of program symbols << >>]. This is necessary because without the program symbol separating the two listings of input variables, namely V T N << V T n, the program will assume that the input command

→V T N V T n

requires six input values while only three are available. The result would have been the generation of an error message and the interruption of the program execution.

To include the subprogram mentioned above in the modified definition of program [p], will require you to use [↵][<< >>] at the beginning and end of the sub-program. Because the program symbols occur in pairs whenever [↵][<< >>] is invoked, you will need to erase the closing program symbol (>>) at the beginning, and the opening program symbol (<<) at the end, of the sub-program.

To erase any character while editing the program, place the cursor to the right of the character to be erased and use the backspace key [↵].

Store the program back into variable p by using [↵][p]. Next, run the program by pressing [p]. Enter values of $V = 0.01_m^3$, $T = 300_K$, and $n = 0.8_mol$, when prompted. Before pressing [ENTER] for input, the stack will look like this:

```
Enter V, T and n:

:V:0.01_m^3
:T:300_K
:n:0.8_mol
```

After execution of the program, the stack will look like this:

```
5:
4:          V:0.01_m^3
3:          T:300_K
2:          n:0.8_mol
1:  p: 200268.24_J/m^3
```

In summary: The common thread in the three examples shown here is the use of tags to identify input and output variables. If we use an input string to get our input values, those values are already pre-tagged and can be easily recall into the stack for output. Use of the →TAG command allows us to identify the output from a program.

Using a message box for output

A message box is a fancier way to present output from a program. The message box command in the HP 49 G is obtained by using [↵][PRG][NXT][OUT][MSGBO]. The message box command requires that the output string to be placed in the box be available in stack level 1. To see the operation of the MSGBOX command try the following exercise:

[↵][“][ALPHA][↵][T][ALPHA][↵][:][1][.][2][↵][_]	Starts string “0:1.2_
[ALPHA][↵][ALPHA][ALPHA][ALPHA][R][A][D][ENTER]	Completes string “0:1.2_rad”
[↵][PRG][NXT][OUT][MSGBO]	Places string in a message box
[OK]	To cancel message box output




You could use a message box for output from a program by using a tagged output, converted to a string, as the output string for MSGBOX. *To convert any tagged result, or any algebraic or non-tagged value, to a string, use the function →STR available at [↵][PRG][TYPE][→STR].*

Example – using a message box for program output

The function [p], from the last example, can be modified to read:

```
<< "Enter V, T and n: " { "↵" :V:↵ :T:↵ :n: " { 2 0 } V } INPUT OBJ→ →V T n <<V T  
n '(8.34451_J/(K*mol))*(n*T/V)' EVAL "p" →TAG →STR MSGBOX>> >>
```

You know the drill:

-  Store the program back into variable p by using [↵][p].
-  Run the program by pressing [p].
-  Enter values of V = 0.01_m^3, T = 300_K, and n = 0.8_mol, when prompted.

As in the earlier version of [p], before pressing [ENTER] for input, the stack will look like this:

```
Enter V, T and n:
:V:0.01_m^3
:T:300_K
:n:0.8_mol
```

The first program output is a message box containing the string:

```
:p:
'200268.24_J/m^3'
```

Press [OK] to cancel message box output. The stack will now look like this:

```
5:
4:
3:          V:0.01_m^3
2:          T:300_K
1:          n:0.8_mol
```

Including input and output in a message box – string concatenation

We could modify the program so that not only the output, but also the input, is included in a message box. For the case of program [p], the modified program will look like:

```
<< "Enter V, T and n: " { "↵" :V:↵ :T:↵ :n: " { 2 0 } V } INPUT OBJ→ →V T n <<V  
→STR "↵ " + T →STR "↵ " + n →STR "↵ " + '(8.34451_J/(K*mol))*(n*T/V)' EVAL "p" →TAG  
→STR + + + MSGBOX>> >>
```

Notice that you need to add the following piece of code after each of the variable names V, T, and n, within the sub-program:

```
→STR "↵ " +
```

To get this piece of code typed in the first time use:

```
[↵][PRG][TYPE] [→STR] [↵]" " [↵][↵] [▶] [+]
```

Because the functions for the TYPE menu remain available in the soft menu keys, for the second and third occurrences of the piece of code (\rightarrow STR " \leftarrow " +) within the sub-program (i.e., after variables T and n, respectively), all you need to use is:




[\rightarrow STR] [\rightarrow] [" "] [\rightarrow] [\leftarrow] [\rightarrow] [+]

You will notice that after typing the keystroke sequence [\rightarrow] [\leftarrow] the result is that a new line is generated in the stack.

The last modification that needs to be included is to type in the plus sign three times after the call to the function at the very end of the sub-program.

Note: The plus sign (+) in this program is used to *concatenate* strings. *Concatenation* is simply the operation of joining individual character strings.

To see the program operating:

-  Store the program back into variable p by using [\leftarrow] [p].
-  Run the program by pressing [p].
-  Enter values of V = 0.01_m^3, T = 300_K, and n = 0.8_mol, when prompted.

As in the earlier version of [p], before pressing [ENTER] for input, the stack will look like this:

```
Enter V, T and n:

:V:0.01_m^3
:T:300_K
:n:0.8_mol
```

The first program output is a message box containing the string:

```
:V:      '.01_m^3'
:T:      '300_K'
:n:      '.8_mol'
:p:
'200268.24_J/m^3'
```

Press [OK] to cancel message box output.

Incorporating units within a program

As you have been able to observe from all the examples for the different versions of program [p] presented in this chapter, attaching units to input values may be a tedious process. You could have the program itself attach those units to the input and output values. We will illustrate these options by modifying yet once more the program [p], as follows.

Recall the contents of program [p] to the stack by using [\rightarrow] [p], and modify them to look like this:

Note: I've separated the program arbitrarily into several lines for easy reading. This is not necessarily the way that the program shows up in the calculator's stack. The sequence of commands is correct, however. Also, recall that the character \leftarrow does not show in the stack, instead it produces a new line.

```
<< "Enter V,T,n [S.I.]: " { "← :V:← :T:← :n: " { 2 0 } V } INPUT OBJ→ →V T n
<<V '1_m^3' * { } + T '1_K' * + n '1_mol' * + EVAL →V T n
<<V "V" →TAG →STR "← " + T "T" →TAG →STR "← " + n "n" →TAG →STR "← " +
'(8.34451_J/(K*mol)) * (n*T/V)' EVAL "p" →TAG →STR + + + MSGBOX>> >> >>
```




This new version of the program includes an additional level of sub-programming (i.e., a third level of program symbols << >>), and some steps using lists, i.e.,

```
V '1_m^3' * { } + T '1_K' * + n '1_mol' * + EVAL →V T n
```

The *interpretation* of this piece of *code* is as follows. (We use input string values of :V:0.01, :T:300, and :n:0.8):

1. V : The value of V, as a tagged input (e.g., V:0.01) is placed in the stack.
2. '1_m^3' : The S.I. units corresponding to V are then placed in stack level 1, the tagged input for V is moved to stack level 2.
3. * : By multiplying the contents of stack levels 1 and 2, we generate a number with units (e.g., 0.01_m^3), but the tag is lost.
4. { } : An empty list is placed on stack level 1, and the previous result is moved to stack level 2.
5. + : The contents of stack level 2 are 'added' to the empty list resulting in the list {0.01_m^3}.
6. T '1_K' * +: : These operations result in the value of T, with its proper units, being 'added' to the list under consideration, resulting in the list being expanded to {0.01_m^3 300_K }
7. n '1_mol' * + : These operations result in the value of n, with its proper units, being 'added' to the list under consideration, resulting in the list being expanded to {0.01_m^3 300_K 0.8_mol }
8. EVAL : This command has the effect of decomposing the list and placing its elements in order in the stack.
9. →V T n : The values of V, T, and n, located respectively in stack levels 3, 2, and 1, are passed on to the next level of sub-programming.

To see this version of the program in action do the following:

-  Store the program back into variable p by using [←][p].
-  Run the program by pressing [p].
-  Enter values of V = 0.01, T = 300, and n = 0.8, when prompted (no units required now).

Before pressing [ENTER] for input, the stack will look like this:

```
Enter V,T,n [S.I.]:
:V:0.01
:T:300
:n:0.8
```

Press [ENTER] to run the program. The output is a message box containing the string:

```
:V:  '.01_m^3'
:T:  '300_K'
:n:  '.8_mol'
:p:
'200268.24_J/m^3'
```

Press [OK] to cancel message box output.

Note: The actual way that the program shows up in the stack when editing it is the following:

```
<<
"Enter V,T,n [S.I.]: "
{
:V:
:T:
:n: " {2.
0.} V } INPUT OBJ→ →
V T n
<<V '1_m^3' * { }
+ T '1_K' * + n '1_
mol' * + EVAL →V T n
<<V "V" →TAG →STR
"
" + T "T" →TAG →STR
"
" + n "n" →TAG →STR
"
" + '(8.34451_J/(K*
mol))* (n*T/V)' EVAL
"p" →TAG →STR + + +
MSGBOX
>>
>>
>>
```

Styling note: You may have noted in all these exercises that unit objects, such as the four objects listed in the message box shown above, keep the quotation marks when transformed into strings. Thus, when shown in a message box, the output may not be aesthetically pleasant. Pure numerical results can easily be translated into strings and thus, results without units will produce better-looking results than those with units. Thus, my recommendation is to use simple tagged output when attaching units to the results, and use message boxes if the output is entirely numerically.

Example - Message box output without units

Let's modify the program [p] once more to eliminate the use of units throughout it. The unit-less program will look like this:

```
<< "Enter V,T,n [S.I.]: " { "␣ :V:␣ :T:␣ :n: " {2 0} V } INPUT OBJ→ →V T n
<<V DTAG {} + T DTAG + n DTAG + EVAL →V T n
<<"V=" V →STR + "␣ " + "T=" T →STR + "␣ " + "n=" n →STR + "␣ " +
'8.34451*n*T/V' EVAL →STR "p=" SWAP + + + + MSGBOX>> >> >>
```

And when run with the input data $V = 0.01$, $T = 300$, and $n = 0.8$, produces the message box output:

V=.01
T=300.
n=.8
p=200268.24

Press [OK] to cancel the message box output.

Relational and logical operators

So far we have worked with sequential programs only. The User RPL language provides statements that allow branching and looping of the program flow. Many of these make decisions based on whether a logical statement is true or not. In this section we present some of the elements used to construct such logical statements, namely, relational and logical operators.

Relational operators

Relational operators are those operators used to compare the relative position of two objects. For example, dealing with real numbers only, relational operators are used to make a statement regarding the relative position of two or more real numbers. Depending on the actual numbers used, such a statement can be true (represented by the numerical value of 1. in the calculator), or false (represented by the numerical value of 0. in the calculator).

The relational operators available for the HP 49 G calculator are:

Operator	Meaning	Example
==	“is equal to”	‘x==2’
≠	“is not equal to”	‘3 ≠ 2’
<	“is less than”	‘m<n’
>	“is greater than”	‘10>a’
≥	“is greater than or equal to”	‘p ≥ q’
≤	“is less than or equal to”	‘7≤12’

All of the operators, except == (which can be created by typing [↵][=][↵][=]), are available in the keyboard as secondary functions in the keys [+/-], [X], and [1/X]. They are also available in the first menu obtained by using:

[←][PRG][TEST].

The examples shown above represent that use relational operators are elementary logical statements that could be true (1.), false (0.), or could simply not be evaluated. To determine whether a logical statement is true or not, place the statement in stack level 1, and press [↵][EVAL]. Examples:

‘2<10’ [↵][EVAL], result: 1. (true)

‘2>10’ [↵][EVAL], result: 0. (false)

In the next example it is assumed that the variable m is not initialized (it has not been given a numerical value):

‘2==m’ [↵][EVAL], result: ‘2==m’

The fact that the result from evaluating the statement is the same original statement indicates that the statement can not be evaluated uniquely.

Logical operators

Logical operators are logical particles that are used to join or modify simple logical statements. The logical operators available in the HP 49 G can be easily accessed through the keystroke sequence:

[\neg][PRG][NXT][TEST][NXT].

The available logical operators are: AND, OR, XOR (exclusive or), NOT, and SAME. The operators will produce results that are true or false, depending on the truth-value of the logical statements affected. The operator NOT (negation) applies to a single logical statements. All of the others apply to two logical statements.

Tabulating all possible combinations of one or two statements together with the resulting value of applying a certain logical operator produces what is called the *truth table of the operator*. The following are truth tables of each of the operators shown above. The symbol between parentheses is the symbol for the particular logical operator used traditionally in mathematical logic. The (logical) variables p and q represent logical statements that are either true (1.) or false (0.):

NOT(\sim , \neg): produces the opposite possibility to that of the statement being negated:

Mathematical logic		HP 49 G implementation	
p	$\sim p$	P	$NOT P$
T	F	1	0
F	T	0	1

AND (\wedge): produces a true result only if both logical statements being linked are true:

Mathematical logic			HP 49 G implementation		
p	q	$p \wedge q$	P	Q	$P AND Q$
T	T	T	1	1	1
T	F	F	1	0	0
F	T	F	0	1	0
F	F	F	0	0	0

OR (\vee): produces a true result as long as at least one of the statements being linked is true:

Mathematical logic			HP 49 G implementation		
p	q	$p \vee q$	P	Q	$P OR Q$
T	T	T	1	1	1
T	F	T	1	0	1
F	T	T	0	1	1
F	F	F	0	0	0

XOR (\diamond): produces a true result only if the two statements being linked have different truth values:

Mathematical logic			HP 49 G implementation		
p	q	$p \diamond q$	P	Q	$P XOR Q$
T	T	F	1	1	0
T	F	T	1	0	1
F	T	T	0	1	1
F	F	F	0	0	0

SAME : this is a non-standard logical operator used to determine if objects in stack levels 1 and 2 are identical. If they are identical, a value of 1. (true) is returned, if not, a value of 0. (false) is returned. For example, try the following exercise:

[\rightarrow]['] [\leftarrow][x^2][2][ENTER]
 [4][ENTER]
 [\leftarrow][PRG][TEST][NXT][SAME]

Enter 'SQ(2)'
 Enter 4
 Result: 0 (false).

Please notice that the use of SAME implies a very strict interpretation of the word "identical." For that reason, SQ(2) is not identical to 4, although they both evaluate to 4.

Program branching

Branching of a program flow implies that the program makes a decision among two or more possible flow paths. The User RPL language provides a number of commands that can be used for program branching. The menus containing these commands are accessed through the keystroke sequence:

[\leftarrow][PRG][BRCH]

This menu shows sub-menus for the program constructs

[IF][CASE][START][FOR][DO][WHILE]

The program constructs IF...THEN..ELSE...END, and CASE...THEN...END will be referred to as program branching construct. The remaining constructs (START, FOR, DO, WHILE) are appropriate for controlling repetitive processing within a program and will be referred to as Program Looping. The latter types of program constructs are presented in more detail in a later section.

What IF...?

In this section we presents examples using the constructs IF...THEN...END and IF...THEN...ELSE...END.

IF...THEN...END

The IF...THEN...END is the simplest of the IF program constructs. The general format of this construct is:

IF logical_statement THEN program_statements END.

The operation of this construct is as follows:

1. Evaluate logical_statement.
2. If logical_statement is true, perform program _statements and continue program flow after the END statement.
3. If logical_statement is false, skip program_statements and continue program flow after the END statement.

To type in the particles IF, THEN, ELSE, and END, use:

[↵][PRG][BRCH][IF].

The functions [IF][THEN][ELSE][END] are available in that menu to be typed selectively by the user. Alternatively, to produce an IF...THEN...END construct directly on the stack, use:

[↵][PRG][BRCH][↵][IF].

This will create the following input in the stack:

```
2:
1:
IF ◀
THEN
END
```

With the cursor ◀ in front of the IF statement prompting the user for the logical statement that will activate the IF construct when the program is executed.

Example: Type in the following program:

```
<< →x << IF 'x<3' THEN 'x^2' EVAL END "Done" MSGBOX >> >>
```

and save it under the name 'f1'.

To type the program in you could use the following:

[↵][<◇>][↵][→] [ALPHA][↵][X]	Start first program level, type in →x
[↵][<◇>][↵][PRG][BRCH][↵][IF]	Start second program level, generate IF.THEN.ELSE
[↵]['] [ALPHA][↵][X] [↵][<] [3] [▼]	Type in logical statement for the IF
[ALPHA][↵][X] [y ^x] [2] [▼][↵][EVAL]	Type in program statements to follow if logical statement true
[↵]["] [ALPHA][D] [ALPHA][↵][O]	Type in "Do
[ALPHA][↵][N] [ALPHA][↵][E] [▶]	Type in ne"
[↵][PRG][NXT][OUT][MSGBO][↵]	Type in MSGBOX
[ENTER]	Enter program in stack
[↵]['] [ALPHA][↵][F][1]	Type in name of program in stack
[STO▶]	Store program

Press [VAR] and verify that variable [f1] is indeed available in your variable menu. Try the operation of the program by using the following input:

[0] [f1]	Results:	Message box:	Done	[OK]	Stack level 1: 0. (i.e., x ²) [↵] (clear stack)
[1][.][2][f1]	Results:	Message box:	Done	[OK]	Stack level 1: 1.44 (i.e., x ²) [↵]
[3][.][5][f1]	Results:	Message box:	Done	[OK]	Stack level 1: empty – no action taken
[1][0][f1]	Results:	Message box:	Done	[OK]	Stack level 1: empty – no action taken

These results confirm the correct operation of the IF...THEN...END construct. The program, as written, calculates the function $f_1(x) = x^2$, if $x < 3$ (and not output otherwise).

IF...THEN...ELSE...END

The IF...THEN...ELSE...END construct permits two alternative program flow paths based on the truth value of the logical_statement. The general format of this construct is:

```
IF logical_statement THEN program_statements_if_true ELSE
    program_statements_if_false END.
```

The operation of this construct is as follows:

1. Evaluate logical_statement.
2. If logical_statement is true, perform program_statements_if_true and continue program flow after the END statement.
3. If logical_statement is false, perform program_statements_if_false and continue program flow after the END statement.

To produce an IF...THEN...ELSE...END construct directly on the stack, use:

[←][PRG][BRCH][→][IF].

This will create the following input in the stack:

1:
IF ←
THEN
ELSE
END

Example: Type in the following program:

```
<< → x << IF 'x<3' THEN 'x^2' ELSE '1-x' END EVAL "Done" MSGBOX >> >>
```

and save it under the name 'f2'.

Try the operation of the program by using the following input:

[0] [f2]	Results:	Message box:	Done	[OK]	Stack level 1: 0. (i.e., x^2) [↵] (clear stack)
[1] [.] [2] [f2]	Results:	Message box:	Done	[OK]	Stack level 1: 1.44 (i.e., x^2) [↵]
[3] [.] [5] [f2]	Results:	Message box:	Done	[OK]	Stack level 1: -2.5 (i.e., $1-x$) [↵]
[1] [0] [f2]	Results:	Message box:	Done	[OK]	Stack level 1: -9 (i.e., $1-x$) [↵]

These results confirm the correct operation of the IF...THEN...ELSE...END construct. The program, as written, calculates the function

$$f_2(x) = \begin{cases} x^2, & \text{if } x < 3 \\ 1 - x, & \text{otherwise} \end{cases}.$$

Note: For this particular case, a valid alternative would have been to use an IFTE function of the form:
 'f2(x) = IFTE(x<3,x^2,1-x)'

Nested IF...THEN...ELSE...END constructs

In most computer programming language where the IF...THEN...ELSE...END construct is available, the general format used for program presentation is the following:

```
IF logical_statement THEN
    program_statements_if_true
ELSE
    program_statements_if_false
END.
```

In designing a calculator program that includes IF constructs, you could start by writing by hand the pseudo-code for the IF constructs as shown above. For example, for program [f2], you could write

```
IF x<3 THEN
    x^2
ELSE
    1-x
END.
```

While this simple construct works fine when your function has only two branches, you may need to nest IF...THEN...ELSE...END constructs to deal with function with three or more branches. For example, consider the function

$$f_3(x) = \left\{ \begin{array}{l} x^2, \text{ if } x < 3 \\ 1-x, \text{ if } 3 \leq x < 5 \\ \sin(x), \text{ if } 5 \leq x < 3\pi \\ \exp(x), \text{ if } 3\pi \leq x < 15 \\ -2, \text{ elsewhere} \end{array} \right\}.$$

Here is a possible way to evaluate this function:

```
IF x<3 THEN
    x^2
ELSE
    IF x<5 THEN
        1-x
    ELSE
```

```

        IF  $x < 3\pi$  THEN
            sin(x)
        ELSE
            IF  $x < 15$  THEN
                exp(x)
            ELSE
                -2
            END
        END
    END
END

```

A complex IF construct like this is called a set of *nested* IF...THEN...ELSE...END constructs. In most computer programming language such set of nested IF constructs is simplified to read:

```

IF  $x < 3$  THEN
     $x^2$ 
ELSE IF  $x < 5$  THEN
    1-x
ELSE IF  $x < 3\pi$  THEN
    sin(x)
ELSE IF  $x < 15$  THEN
    exp(x)
ELSE
    -2
END

```

This construct is known as an IF...THEN...ELSEIF construct. Such a construct is, however, not possible in User RPL language. Therefore, you will have to type in the earlier construct with the four END statements as shown.

A possible way to evaluate $f_3(x)$, based on the nested IF construct shown above, is to write the program:

```

<< → x << IF 'x<3' THEN 'x^2' ELSE IF 'x<5' THEN '1-x' ELSE IF 'x<3*π' THEN 'SIN(x)'
ELSE IF 'x<15' THEN 'EXP(x)' ELSE -2 END END END END EVAL >> >>

```

Store the program in variable [f3] and try the following evaluations:

1.5	[f3]	Result: 2.25 (i.e., x^2)
2.5	[f3]	Result: 6.25 (i.e., x^2)
4.2	[f3]	Result: -3.2 (i.e., 1-x)
5.6	[f3]	Result: -0.631266... (i.e., sin(x), with x in radians)
12	[f3]	Result: 162754.791419 (i.e., exp(x))
23	[f3]	Result: -2. (i.e., -2)

Just in CASE ...

The CASE construct can be used to code several possible program flux paths, as in the case of the nested IF constructs presented earlier. The general format of this construct is as follows:

```

CASE
Logical_statement1 THEN program_statements1 END
Logical_statement2 THEN program_statements2 END
.
.
.
Logical_statement THEN program_statements END
Default_program_statements (optional)
END



```

When evaluating this construct, the program tests each of the *logical_statements* until it finds one that is true. The program executes the corresponding *program_statements*, and passes program flow to the statement following the END statement.

The CASE, THEN, and END statements are available for selective typing by using

[↵][PRG][BRCH][CASE].

If you are in the BRCH menu, i.e., ([↵][PRG][BRCH]) you can use the following shortcuts to type in your CASE construct (The location of the cursor is indicated by the symbol ◀):

 [↵][CASE]	: Starts the case construct providing the prompts: CASE ◀ THEN END END
 [↵][CASE]	: Completes a CASE line by adding the particles THEN ◀ END

Example – program $f_3(x)$ using the CASE statement

The function is defined by the following 5 expressions:

$$f_3(x) = \begin{cases} x^2, & \text{if } x < 3 \\ 1 - x, & \text{if } 3 \leq x < 5 \\ \sin(x), & \text{if } 5 \leq x < 3\pi \\ \exp(x), & \text{if } 3\pi \leq x < 15 \\ -2, & \text{elsewhere} \end{cases}.$$

Using the CASE statement in User RPL language we can code this function as:

```

<< → x << CASE 'x<3' THEN 'x^2' END 'x<5' THEN '1-x' END 'x<3*π' THEN 'SIN(x)' END 'x<15'
THEN 'EXP(x)' END -2 END EVAL >> >>

```

Store the program into a variable called [f3c]. Then, try the following exercises:

1.5	[f3c]	Result: 2.25 (i.e., x^2)
2.5	[f3c]	Result: 6.25 (i.e., x^2)
4.2	[f3c]	Result: -3.2 (i.e., $1-x$)
5.6	[f3c]	Result: -0.631266... (i.e., $\sin(x)$, with x in radians)
12	[f3c]	Result: 162754.791419 (i.e., $\exp(x)$)
23	[f3c]	Result: -2. (i.e., -2)

As you can see, f3c produces exactly the same results as f3. The only difference in the programs is the branching constructs used. For the case of function $f_3(x)$, which requires five expressions for its definition, the CASE construct may be easier to code than a number of nested IF...THEN...ELSE...END constructs.

Program loops

Program loops are constructs that permit the program the execution of a number of statements repeatedly. For example, suppose that you want to calculate the summation of the square of the integer numbers from 0 to n, i.e.,

$$S = \sum_{k=0}^n k^2.$$


Well, with the HP 49 G calculator, all that you have to do is use the $\leftarrow[\Sigma]$ key within the equation editor and load the limits and expression for the summation (examples of summations were presented in Chapter 6). However, in order to illustrate the use of programming loops, we will calculate this summation with our own User RPL codes. There are four different commands that can be used to code a program loop in User RPL, these are START, FOR, DO, and WHILE. The commands START and FOR use an index or counter to determine how many times the loop is executed. The commands DO and WHILE rely on a logical statement to decide when to terminate a loop execution. Operation of the loop commands is described in detail in the following sections.

START

The START construct uses a couple of values of an index to repeat a number of statements. There are two versions of the START construct: START...NEXT and START ... STEP. The START...NEXT version is used when the index increment is equal to 1, and the START...STEP version is used when the index increment is determined by the user.

Commands involved in the START construct are available through: $\leftarrow[\text{PRG}][\text{BRCH}][\text{START}]$.

Within the BRCH menu ($\leftarrow[\text{PRG}][\text{BRCH}]$) the following keystrokes are available to generate START constructs (the symbol indicates cursor position):

 $\leftarrow[\text{START}]$: Starts the START...NEXT construct: START \leftarrow NEXT

 $\leftarrow[\text{START}]$: Starts the START...STEP construct: START \leftarrow STEP

START...NEXT

The general form of this statement is:

```
start_value end_value START program_statements NEXT
```

Because for this case the increment is 1, in order for the loop to end you should ensure that $\text{start_value} < \text{end_value}$. Otherwise you will produce what is called an *infinite (never-ending) loop*.

Example – calculating of the summation S defined above

The START...NEXT construct contains an index whose value is inaccessible to the user. Since for the calculation of the sum the index itself (k, in this case) is needed, we must create our own index, k, that we will increment within the loop each time the loop is executed. A possible implementation for the calculation of S is the program:

```
<< 0. DUP → n S k << 0. n START k SQ S + 1. k + 'k' STO 'S' STO NEXT S "S" TAG >>
>>
```

Type the program in, and save it in a variable called [S1].

Here is a brief explanation of how the program works:

1. This program needs an integer number as input. Thus, before execution, that number (n) is in stack level 1. The program is then executed.
2. A zero is entered, moving n to stack level 2.
3. The command DUP, which can be typed in as [ALPHA][ALPHA][D][U][P][ALPHA], copies the contents of stack level 1, moves all the stack levels upwards, and places the copy just made in stack level 1. Thus, after DUP is executed, n is in stack level 3, and zeroes fill stack levels 1 and 2.
4. The piece of code $\rightarrow n \ S \ k$ stores the values of n, 0, and 0, respectively into local variables n, S, k. We say that the variables n, S, and k have been *initialized* (S and k to zero, n to whatever value the user chooses).
5. The piece of code $0. \ n \ START$ identifies a START loop whose index will take values of 0, 1, 2, ..., n
6. The sum S is incremented by k^2 in the piece of code that reads: $k \ SQ \ S +$
7. The index k is incremented by 1 in the piece of code that reads: $1. \ k +$
8. At this point, the updated values of S and k are available in stack levels 2 and 1, respectively. The piece of code 'k' STO stores the value from stack level 1 into local variable k. The updated value of S now occupies stack level 1.
9. The piece of code 'S' STO stores the value from stack level 1 into local variable k. The stack is now empty.
10. The particle NEXT increases the index by one and sends the control to the beginning of the loop (step 6).
11. The loop is repeated until the loop index reaches the maximum value, n.
12. The last part of the program recalls the last value of S (the summation), tags it, and places it in stack level 1 to be viewed by the user as the program output.

To see the program in action, step by step, you can use the debugger as follows (use n = 2). Let SL1 mean stack level 1:

[VAR][2][↩][' '][S1][ENTER]	Place a 2 in level 2, and the program name, 'S1', in level 1
[↩][PRG][NXT][NXT][RUN][DEBUG]	Start the debugger. SL1 = 2.
[SST↓]	SL1 = 0., SL2 = 2.
[SST↓]	SL1 = 0., SL2 = 0., SL3 = 2. (DUP)
[SST↓]	Empty stack ($\rightarrow n \ S \ k$)
[SST↓]	Empty stack (\ll - start subprogram)
[SST↓]	SL1 = 0., (start value of loop index)
[SST↓]	SL1 = 2.(n), SL2 = 0. (end value of loop index)
[SST↓]	Empty stack (START – beginning of loop)

--- loop execution number 1 for k = 0	
[SST↓]	SL1 = 0. (k)
[SST↓]	SL1 = 0. ($SQ(k) = k^2$)
[SST↓]	SL1 = 0.(S), SL2 = 0. (k^2)

[SST↓]	SL1 = 0. ($S + k^2$)
[SST↓]	SL1 = 1., SL2 = 0. ($S + k^2$)
[SST↓]	SL1 = 0.(k), SL2 = 1., SL3 = 0. ($S + k^2$)
[SST↓]	SL1 = 1.(k+1), SL2 = 0. ($S + k^2$)
[SST↓]	SL1 = 'k', SL2 = 1., SL3 = 0. ($S + k^2$)
[SST↓]	SL1 = 0. ($S + k^2$) [Stores value of SL2 = 1, into SL1 = 'k']
[SST↓]	SL1 = 'S', SL2 = 0. ($S + k^2$)
[SST↓]	Empty stack [Stores value of SL2 = 0, into SL1 = 'S']
[SST↓]	Empty stack (NEXT – end of loop)

--- loop execution number 2 for k = 1

[SST↓]	SL1 = 1. (k)
[SST↓]	SL1 = 1. ($SQ(k) = k^2$)
[SST↓]	SL1 = 0.(S), SL2 = 1. (k^2)
[SST↓]	SL1 = 1. ($S + k^2$)
[SST↓]	SL1 = 1., SL2 = 1. ($S + k^2$)
[SST↓]	SL1 = 1.(k), SL2 = 1., SL3 = 1. ($S + k^2$)
[SST↓]	SL1 = 2.(k+1), SL2 = 1. ($S + k^2$)
[SST↓]	SL1 = 'k', SL2 = 2., SL3 = 1. ($S + k^2$)
[SST↓]	SL1 = 1. ($S + k^2$) [Stores value of SL2 = 2, into SL1 = 'k']
[SST↓]	SL1 = 'S', SL2 = 1. ($S + k^2$)
[SST↓]	Empty stack [Stores value of SL2 = 1, into SL1 = 'S']
[SST↓]	Empty stack (NEXT – end of loop)

--- loop execution number 3 for k = 2

[SST↓]	SL1 = 2. (k)
[SST↓]	SL1 = 4. ($SQ(k) = k^2$)
[SST↓]	SL1 = 1.(S), SL2 = 4. (k^2)
[SST↓]	SL1 = 5. ($S + k^2$)
[SST↓]	SL1 = 1., SL2 = 5. ($S + k^2$)
[SST↓]	SL1 = 2.(k), SL2 = 1., SL3 = 5. ($S + k^2$)
[SST↓]	SL1 = 3.(k+1), SL2 = 5. ($S + k^2$)
[SST↓]	SL1 = 'k', SL2 = 3., SL3 = 5. ($S + k^2$)
[SST↓]	SL1 = 5. ($S + k^2$) [Stores value of SL2 = 3, into SL1 = 'k']
[SST↓]	SL1 = 'S', SL2 = 5. ($S + k^2$)
[SST↓]	Empty stack [Stores value of SL2 = 0, into SL1 = 'S']
[SST↓]	Empty stack (NEXT – end of loop)

--- for n = 2, the loop index is exhausted and control is passed to the statement following NEXT

[SST↓]	SL1 = 5 (S is recalled to the stack)
[SST↓]	SL1 = "S", SL2 = 5 ("S" is placed in the stack)
[SST↓]	SL1 = S:5 (tagging output value)
[SST↓]	SL1 = S:5 (leaving sub-program >>)
[SST↓]	SL1 = S:5 (leaving main program >>)

The step-by-step listing is finished. The result of running program [S1] with n = 2, is S:5.

Check also the following results:

[VAR]			
[3][S1]	Result: S:14	[4][S1]	Result: S:30
[5][S1]	Result: S:55	[8][S1]	Result: S:204
[1][0][S1]	Result: S:385	[2][0][S1]	Result: S:2870
[3][0][S1]	Result: S:9455	[1][0][0][S1]	Result: S:338350

START...STEP

The general form of this statement is:

```
start_value end_value START program_statements increment NEXT
```

The start_value, end_value, and increment of the loop index can be positive or negative quantities. For increment > 0, execution occurs as long as the index is less than or equal to end_value. For increment < 0, execution occurs as long as the index is greater than or equal to end_value.

Example – generating a list of values

Suppose that you want to generate a list of values of x from x = 0.5 to x = 6.5 in increments of 0.5. You can write the following program:



```
<< → xs xe dx << xs DUP xe START DUP dx + dx STEP DROP xe xs - dx / ABS 1. +  
→LIST >> >>
```

and store it in variable [GLIST].

In this program, xs = starting value of the loop, xe = ending value of the loop, dx = increment value for loop. The program places values of xs, xs+dx, xs+2·dx, xs+3·dx, ... in the stack. Then, it calculates the number of elements generated using the piece of code:

```
xe xs - dx / ABS 1. +
```

Finally, the program puts together a list with the elements placed in the stack.

-  Check out that the program call 0.5 2.5 0.5 [GLIST] produces the list {0.5 1. 1.5 2. 2.5}.
-  To see step-by-step operation use the program DBUG for a short list, for example:

[VAR][1][SPC][1][.][5][SPC][.][5]	Enter parameters 1 1.5 0.5
[↵]['][GLIST][ENTER]	Enter the program name, 'S1', in level 1
[↵][PRG][NXT][NXT][RUN][DEBUG]	Start the debugger.



Use [SST↓] to step into the program and see the detailed operation of each command.

FOR

As in the case of the START command, the FOR command has two variations: the FOR...NEXT construct, for loop index increments of 1, and the FOR...STEP construct, for loop index increments selected by the user. Unlike the START command, however, the FOR command does require that we provide a name for the loop index (e.g., j, k, n). We need not to worry about incrementing the index ourselves, as done in the examples using START. The value corresponding to the index is available for calculations.

Commands involved in the FOR construct are available through: [↵][PRG][BRCH][FOR].

Within the BRCH menu ([↵][PRG][BRCH]) the following keystrokes are available to generate FOR constructs (the symbol indicates cursor position):

-  [↵][FOR] : Starts the FOR...NEXT construct: FOR ◀ NEXT
-  [↵][FOR] : Starts the FOR...STEP construct: FOR ◀ STEP

FOR...NEXT

The general form of this statement is:

```
start_value end_value FOR loop_index program_statements NEXT
```

To avoid an infinite loop, make sure that `start_value < end_value`.

Example – calculate the summation S using a FOR...NEXT construct

The following program calculates the summation

$$S = \sum_{k=0}^n k^2.$$

Using a FOR...NEXT loop:

```
<< 0. →n S << 0. n FOR k k SQ S + 'S' STO NEXT S "S" TAG >> >>
```

Store this program in a variable [S2]. Verify the following exercises:

[VAR]

[3][S2] *Result:* S:14

[4][S2] *Result:* S:30

[5][S2] *Result:* S:55

[8][S2] *Result:* S:204

[1][0][S2] *Result:* S:385

[2][0][S2] *Result:* S:2870

[3][0][S2] *Result:* S:9455

[1][0][0][S2] *Result:* S:338350

You may have noticed that the program is much simpler than the one stored in [S1]. There is no need to initialize k , or to increment k within the program. The program itself takes care of producing such increments.

FOR...STEP

The general form of this statement is:

```
start_value end_value FOR loop_index program_statements increment STEP
```

The `start_value`, `end_value`, and `increment` of the loop index can be positive or negative quantities. For `increment > 0`, execution occurs as long as the index is less than or equal to `end_value`. For `increment < 0`, execution occurs as long as the index is greater than or equal to `end_value`.

Example – generate a list of numbers using a FOR...STEP construct

Type in the program:

```
<< → xs xe dx << xe xs - dx / ABS 1. + → n << xs xe FOR x x dx STEP n →LIST >>  
>> >>
```

and store it in variable [GLIS2].

- ✚ Check out that the program call 0.5 2.5 0.5 [GLIS2] produces the list {0.5 1. 1.5 2. 2.5}.
- ✚ To see step-by-step operation use the program DBUG for a short list, for example:

[VAR][1][SPC][1][.][5][SPC][.][5]	Enter parameters 1 1.5 0.5
[↵]['[]][GLIS2][ENTER]	Enter the program name, 'S1', in level 1
[←][PRG][NXT][NXT][RUN][DEBUG]	Start the debugger.

Use [SST↓] to step into the program and see the detailed operation of each command.

DO

The general structure of this command is:

```
DO program_statements UNTIL logical_statement END
```

The DO particle starts an indefinite loop repeating the `program_statements` and checking for the value of `logical_statement` at the end of each repetition. The program sends the control to the statement following END when `logical_statement` becomes 0 (false). The `logical_statement` must contain the value of an index whose value is changed in the `program_statements`.

Example 1 – calculate the summation S using a DO...UNTIL...END construct

The following program calculates the summation

$$S = \sum_{k=0}^n k^2.$$

Using a DO...UNTIL...END loop:

```
<< 0. →n S << DO n SQ S + 'S' STO n 1 - 'n' STO UNTIL 'n<0' END S "S" TAG >> >>
```

Store this program in a variable [S3]. Verify the following exercises:

[VAR]			
[3][S3]	Result: S:14	[4][S3]	Result: S:30
[5][S3]	Result: S:55	[8][S3]	Result: S:204
[1][0][S3]	Result: S:385	[2][0][S3]	Result: S:2870
[3][0][S3]	Result: S:9455	[1][0][0][S3]	Result: S:338350

Example 2 – generate a list using a DO...UNTIL...END construct

Type in the following program

```
<< → xs xe dx << xe xs - dx / ABS 1. + xs → n x << xs DO 'x+dx' EVAL DUP 'x'
STO UNTIL 'x xe' END n →LIST >> >> >>
```

and store it in variable [GLIS3].

- ✚ Check out that the program call 0.5 2.5 0.5 [GLIS3] produces the list {0.5 1. 1.5 2. 2.5}.
- ✚ To see step-by-step operation use the program DBUG for a short list, for example:

[VAR][1][SPC][1][.][5][SPC][.][5]	Enter parameters 1 1.5 0.5
[↵]['[GLIS2][ENTER]	Enter the program name, 'S1', in level 1
[←][PRG][NXT][NXT][RUN][DEBUG]	Start the debugger.

Use [SST↓] to step into the program and see the detailed operation of each command.

WHILE

The general structure of this command is:

```
WHILE logical_statement REPEAT program_statements END
```

The WHILE particle checks that whether the logical_statement is true and, if so, repeats the program_statements. If not, program control is passed to the statement right after END. The program_statements must include a loop index that gets modified before the logical_statement is checked at the beginning of the next repetition.

Example – calculate the summation S using a WHILE...REPEAT...END construct

The following program calculates the summation

$$S = \sum_{k=0}^n k^2.$$

Using a WHILE...REPEAT...END loop:

```
<< 0. →n S << WHILE 'n≥0' REPEAT n SQ S + 'S' STO n 1 - 'n' STO END S "S" TAG >> >>
```

Store this program in a variable [S4]. Verify the following exercises:

[VAR]			
[3][S4]	Result: S:14	[4][S4]	Result: S:30
[5][S4]	Result: S:55	[8][S4]	Result: S:204
[1][0][S4]	Result: S:385	[2][0][S4]	Result: S:2870
[3][0][S4]	Result: S:9455	[1][0][0][S4]	Result: S:338350

Example 2 – generate a list using a WHILE...REPEAT...END construct

Type in the following program

```
<< → xs xe dx << xe xs - dx / ABS 1. + xs → n x << xs WHILE 'x<xe' REPEAT 'x+dx'  
EVAL DUP 'x' STO END n →LIST >> >> >>
```

and store it in variable [GLIS3].

- ✚ Check out that the program call 0.5 2.5 0.5 [GLIS3] produces the list {0.5 1. 1.5 2. 2.5}.
- ✚ To see step-by-step operation use the program DBUG for a short list, for example:

[VAR][1][SPC][1][.][5][SPC][.][5]	Enter parameters 1 1.5 0.5
-----------------------------------	----------------------------

[↵]['][GLIS2][ENTER]

Enter the program name, 'S1', in level 1

[↶][PRG][NXT][NXT][RUN][DEBUG]

Start the debugger.

Use [SST↓] to step into the program and see the detailed operation of each command.

Procedural programming and object-oriented programming

In this chapter we have covered a number of relatively simple programming examples that include the basic ideas of sequential, branching, and looping programs. This type of programming in which the programmer is controlling the program flow at an elementary level is called *procedural programming*.

In the last decade or so, there has been an emphasis in the development and use of *object-oriented* programming. In this approach the user takes advantage of pre-programmed functions (objects) for which he or she provides the appropriate input, and from which he or she receives a pre-specified type of output.

Most of the functions provided by the HP 49 G calculator emphasize object-oriented programming (OOP). For example, to solve a quadratic equation in the HP 49 G calculator, you need to enter a quadratic expression, the variable to solve for, and then use the function QUAD. The user is not concerned with the details of how the roots of the equation were calculated. All that he or she cares about is that the proper type of input is provided to the function or object, and that the calculator will return the desired output.

Concluding remarks on programming

We emphasize once again that the examples and techniques shown in this chapter are very elementary, and that it would be almost impossible to include every aspect of the art of programming the HP 49 G calculator in this chapter. Other documents at InfoClearinghouse.com discuss the utilization of the calculator functions in specific mathematical problems, such as algebra, differential and integral calculus, linear algebra, vector calculus, etc. Additional programming examples will be provided as we discuss those subjects. In the meantime, the user is encouraged to explore routine calculations that can be easily addressed through calculator programs.

REFERENCES (for all HP49 documents at InfoClearinghouse.com)

- Devlin, Keith, 1998, "The Language of Mathematics," W.H. Freeman and Company, New York.
- Farlow, Stanley J., 1982, "Partial Differential Equations for Scientists and Engineers," Dover Publications Inc., New York.
- Friedman, B., 1956, "Principles and Techniques of Applied Mathematics," (reissued 1990), Dover Publications Inc., New York.
- Gullberg, J., 1997, "Mathematics – From the Birth of Numbers," W. W. Norton & Company, New York.
- Harris, J.W., and H. Stocker, 1998, "Handbook of Mathematics and Computational Science," Springer, New York.
- Heath, M. T., 1997, "Scientific Computing: An Introductory Survey," WCB McGraw-Hill, Boston, Mass.
- Hewlett Packard Co., 1999, HP 49 G GRAPHING CALCULATOR USER'S GUIDE.
- Hewlett Packard Co., 2000, HP 49 G GRAPHING CALCULATOR ADVANCED USER'S GUIDE
- Kottegoda, N. T., and R. Rosso, 1997, "Probability, Statistics, and Reliability for Civil and Environmental Engineers," The Mc-Graw Hill Companies, Inc., New York.
- Kreysig, E., 1983, "Advanced Engineering Mathematics – Fifth Edition," John Wiley & Sons, New York.
- Newland, D.E., 1993, "An Introduction to Random Vibrations, Spectral & Wavelet Analysis – Third Edition," Longman Scientific and Technical, New York.
- Tinker, M. and R. Lambourne, 2000, "Further Mathematics for the Physical Sciences," John Wiley & Sons, LTD., Chichester, U.K.