Chapter 10

Introduction to programming

Contents

Introduction

This chapter describes how to create and run programs on the HP 49G. The HP 49G has a rich programming environment. Programs can range in complexity from a simple task such as performing a sequence of arithmetic operations, to a complex process that requests input, performs extensive processing, and outputs results in a graphical format.

This chapter focuses on creating and running programs in algebraic mode only. See the *Advanced User's Guide* for information on creating and using programs in RPN mode. See the *Pocket Guide* for details of the programming commands that are available.

Getting started

This section contains an example of how to create a simple program to calculate the hypotenuse of a right-angle triangle, using Pythagoras' theorem. When you start the program, you supply the known side lengths as arguments.

This program is an example of a simple algebraic calculation using the arguments that you specify. Within the program, the calculation is enclosed in single quotes (→ ') to delimit it as an algebraic object. If you need to do extensive processing involving looping and branching, you use a nested procedure (→ «») to delimit the procedure from the arguments.

The program does the following:

- It collects the known side lengths as arguments and stores them as local variables, that is, variables that exist only while the program is running.
- It uses the variables to calculate the length of the hypotenuse, and returns the result to the history.

Create the program as follows:

1. Put the program delimiters on the command line.
   → «» « »
2. Define the two local variables to accept the arguments for the side length.
   → → ALPHA A SPC ALPHA B SPC « → A B »

3. Define the equation to calculate the hypotenuse.

   Note that you need to use ' to enclose the equation and separate it from the definition of the arguments.

   ▶ ' √x ◀ () ALPHA A y^x 2

   + ALPHA B y^x 2 « → A B '√(A^2+B^2)' »

4. Move the cursor out of the program and specify that you want to store the program as "PYTH".

   ▶ ▼ STO▶ ALPHA ALPHA PYTH « →A B '√(A^2+B^2)' » ▶ PYTH

5. Press ENTER to store the program.

   ENTER

When you run the program, you specify the lengths of the sides as arguments to the program. For example, to run the program to calculate the hypotenuse of a right angle triangle with sides of 3 and 4 units:

1. Display a list of the variables in the directory.

   VAR

2. Press the function key that corresponds to your program. The program name is inserted on the command line. Press ◀ () to insert parentheses after the program name.
3. Enter your arguments, separated by a ▶ , , between the parentheses.

   3 ▶ , 4

4. Press ENTER to calculate the hypotenuse.

   ENTER

   The result is returned to history.

Creating, saving, and running a program

A program is an object that you can store in a variable. That is, you create a program, assign it a name and save it in a directory.

- To create a program, press ▶ «». The program delimiters appear on the command line ready for you to enter code, and the PRG annunciator appears at the top of the screen to indicate that you are in program mode.

  Use the keyboard functions and keys, and select commands from the programming menu, to create your program. As you select function keys and operator keys, the functions and operations appear in your program.

Use **;** to separate functions and calculations within a nested procedure. To enter **;**, press and hold down ▶, and press .SPC.

For readability, you can use ▶↵ to add line breaks.

For details on editing a program—for example cutting, copying, and pasting code—see See "Editing the command line" on page 2-13.

- To save your program:
  a. Press ▶▼ to move the cursor past the end of the program.
  b. Press STO▶ to insert the ▶ symbol after the program.
  c. Enter a name for the program, and press ENTER.
- To run a program:
  a. Access the directory where the program resides and either enter the program name on the command line, or press VAR and select it from the function-key menu.

     The program name should now be on the command line.
  b. Press ◀( ) to insert parentheses after the program name.
  c. Enter the argument or arguments separated by a ▶, , and press ENTER.

The programming menu

The programming menu contains the commands you can use in a program. Select a category to display the available commands in that category. From the menu, you select commands to include in your program. The programming menu is a typing aid only. You need to know the syntax of the commands, and how to use them in your program. See the pocket guide for details of programming commands and their syntax.

Examining the programming menu is a good way to get an idea of the types of programming operations that are available on the HP 49G.

To display the programming menu, press ◀PRG.

Algebraic and RPN modes

In RPN mode, the HP 49G makes extensive use of the stack. When developing programs in RPN mode, you use the stack to:

- provide the data that your program uses
- construct the commands that your program uses
- hold the output that your program generates.

In algebraic mode, the stack is not available. You use other methods to build your program and to pass data to it.

Using functions that require arguments

When using a function that requires arguments:

- In RPN mode, you place the arguments on the stack before calling the function.
- In algebraic mode, you supply the arguments, enclosed in parentheses, after the function call.

For example, you can use the INPUT command to prompt for data. The following code segments demonstrate how to use the INPUT command to collect data in both RPN and algebraic modes.

- In RPN mode, the following code segment prompts for input, collects the data as a string and converts it to a number. At the end of the process, the data is on level 1 of the stack:

  ```
  « "ENTER A NUMBER"
  ""
  INPUT
  OBJ→ »
  ```

- In Algebraic mode, the following code segment performs the same operation. At the end of this process, the data is stored in a global variable, NUM1, ready for use in the program.

  Note that, since you are using a global rather than a local variable, you can follow the variable declaration with a function.

  ```
  « INPUT ("ENTER A NUMBER","") ▶ NUM1;
  OBJ→(NUM1) ▶ NUM1 »
  ```

Handling data

This section briefly describes how you can supply data to your programs, and how you can output data that your programs produce.

Input data

You can use one of the following methods to specify the data that you want your program to use:

- as arguments when you run the program
- as variables that you create in memory before you run the program
- by prompting for input as the program runs.
  - See "Using functions that require arguments" on page 10-5 for an example of using the INPUT function to prompt for data.
  - When you use a function such as INPUT to collect numeric data while the program runs, the data is returned as a string. You need to convert it to a number using a function such as OBJ→.

Output data

Data that is output in algebraic mode is written to the history. Note the following points regarding output:

- When the program completes, the history displays the last output only. This is displayed at level 1. To retain outputs created during processing, you can write the output to a global variable or variables as the program progresses. This method gives you the flexibility to format the output, and to add comments to improve clarity.
- Some functions return multiple values. For these functions, values are written to a list. Unless you output to a variable, the list appears on the history.

How a program flows

HP 49G programs have one entrance point—at the beginning of the program—and one exit point—at the end of the program. There is no command such as GOTO that you can use to jump to a point within a program. Within a program, you use looping and branching structures such as `IF THEN` to control the order of operations. See "Conditional and looping structures" on page 10-13 for details.

You can run other programs from within your programs. In this manner, you can create modular programs. For example, you could create three discrete component programs named INPUT, PROCESSING, and OUTPUT. You could then create a master program that runs each of these components in turn, as follows:

```
« INPUT PROCESSING OUTPUT »
```

Nested procedures

If you use local variables to collect input arguments, you need to use nested procedures if you want to perform branching and looping. You cannot perform branching and looping from within an algebraic object.

To insert a new nested procedure in your code, press ↱ (<<>>) to insert the delimiters. Enter the procedure code between the delimiters.

For example, in the following programming segment, the input arguments are assigned to variables A and B. The algebraic object, a calculation that adds the variables, needs to be enclosed in single quotes as it immediately follows the local variable definition. This example returns the sum of A and B to the history.

```
« → A B 'A+B' »
```

In the following programming segment you use a nested procedure, as the processing involves more than a simple calculation. This example compares A and B, and carries out calculations based on the comparison. The results of the calculations are stored in global variables C and D.

```
« → A B
   « IF A>B
      THEN A-B ▶ C; A^2-B^2 ▶ D
         ELSE B-A ▶ C
      END
   »
»
```

Note that within a nested procedure, you need to use ; to separate calculations. To insert a ; character, press and hold ↱ and press (SPC).

Working with variables

You use variables to hold data within your programs. There are two types of variables within the HP 49G programming environment.

- You create **local variables** within your program. For example, local variables hold the values set by the arguments you use when you call the program.

  A program can only access a local variable inside the nested procedure where it was created, and any nested procedures that it contains.
- You can create **global variables** in a program or you can use existing global variables. See chapter 7, "Storing objects" for details on how to create global variables. Note the following points:
  - Global variables are available anywhere within a program.
  - To remove a global variable using code, use the PURGE command.
  - If you use global variables in your program, they must be located in the same directory, or higher, as the program.

Using local variables

There are some constraints with local variables that you need to be aware of. These are as follows:

- Immediately after a local variable declaration, the program code must contain either:
  - an algebraic calculation enclosed in single quotes
  - a nested procedure enclosed by « ».
- A local variable is available in the nested procedure where it was created, and all nested procedures that it contains.
- You can create a local variable with the same name as an existing global variable (that is, a variable in the same directory or higher as the program). Commands that use the variable name will use the local variable value rather than the global variable value.

Setting variables

You generally set variables to inputs or to the results of processes and calculations that your program performs. You can use local variables to store intermediate results that you want to re-use in subsequent nested procedures within your program. Use global variables to store data for wider access.

Setting local variables to hold input arguments

1. On the command line, position the cursor immediately to the right of the opening « symbol.
2. Press ⟦→⟧ to insert the → symbol.
3. Enter a local variable name for each input argument your program uses, separating each with a ⟦SPC⟧.

For example, if your program uses two arguments, and you want to set the value of these arguments to local variables A and B, the beginning of your program would appear as follows:

⟦→⟧⟦«»⟧⟦→⟧⟦ALPHA⟧A⟦SPC⟧⟦ALPHA⟧B

```
« → A B
```

Setting a local variable to a value

After the value, press ⟦→⟧ to insert the ÷ symbol, and enter the local variable name.

For example, to set local variable G to hold 9.81, the acceleration of gravity, you create the variable as follows:

⟦→⟧⟦«»⟧ 9.81⟦→⟧⟦ALPHA⟧G

```
« 9.81 → G »
```

The following example:

- accepts an input argument
- creates the local variable G
- multiplies it by the argument, and places the result on the history.

```
« → A
 « 9.81 → G
  « A*G »
 »
»
```

In the following example, the `A*G` calculation does not recognize the local variable G as 9.81, as it is outside the nested procedure where the variable was declared. The `A+G` calculation recognizes G as 9.81

```
« → A
 « 9.81 → G 'A+G' »
A*G
»
```

Setting a local variable to the result of a calculation

The following program segment demonstrates how to set a local variable to the result of a calculation, and to use the result in a subsequent calculation. The program accepts two input arguments and uses these in the calculations.

1. On the command line, insert the program delimiters and specify the local variables to hold the input arguments.
   ⮠ « » ⮠ → ALPHA A SPC ALPHA B SPC

   « → A B

   »

2. Start a new nested procedure and define the initial calculation.
   ⮠ « » ALPHA A + ALPHA B

   « → A B

   « A+B

   »

   »

3. Store the results of the calculation to local variable C.
   ⮠ → ALPHA C

   « → A B

   « A+B → C

   »

   »

4. Open a new nested procedure and enter a calculation that uses the result of the initial calculation.
   ⮠ « » ALPHA C + √ ⮌ ( ) ALPHA A − ALPHA B
   ⮠ → ALPHA C

   « → A B

   « A+B → C

   « C+√(A-B)

   »

   »

   »

Using global variables

You can use existing global variables in your programs. Global variables are different to local variables in the following ways:

- Global variables are available to the entire program, independent of nested procedures.
- Unlike local variables, you can create more than one global variable in a nested procedure.

Within a program, you use the STO▶ key to define a global variable. The STO▶ key produces a ▶ symbol on the command line.

Example

The following program demonstrates the use of a global variable to hold the data a program uses, and to hold the output it produces. It performs the following tasks:

- It accepts an input argument and calculates its percentage of a value in the global variable "TOTL". You create TOTL before you run the program.
- It stores the result into another global variable, "RESLT1".
- It converts the numeric result to a string and adds "%" for readability.

To create the program, perform the following:

1. Insert the program delimiters onto the command line and define the input variable.

   ↱ « » ↱ → ALPHA A

   « → A

   »

2. Create a new nested procedure.

   ↱ « »

   « → A

   «

   »

   »

3. Enter the percentage calculation.
   ◀ ( ) ALPHA A ÷ ALPHA ALPHA TOTL ALPHA ▶ × 100

   ```
   « → A
   « (A/TOTL)*100
   »
   »
   ```

4. Store the results into the global variable “RESLT1”. Note that after the calculation, you need to insert a ; to delimit the algebraic commands (hold down ▷ and press SPC).
   STO▶ ALPHA ALPHA RESLT1 ALPHA ▷ SPC

   ```
   « → A
   « (A/TOTL)*100 ▶ RESLT1;
   »
   »
   ```

5. Add “%” and save the resulting string back into RESLT1. Note the following:
   - To insert the % symbol, use the Characters tool (▷ CHARS) or press ALPHA ◀ 1.
   - When you add a string to a number, the resulting value is a string. You do not need to convert the number.

   ALPHA ALPHA RESLT1 ALPHA + " " SPC ▷ CHARS % ENTER ▶ STO▶ ALPHA ALPHA RESLT1

   ```
   « → A
   « (A/TOTL)*100 ▶ RESLT1 ;
   RESLT1+" %" ▶ RESLT1
   »
   »
   ```

Before you run this program, create a global variable named “TOTL” and assign a number to it.

Looping and branching

This section introduces the use of conditional branching and looping within a program. Conditional structures evaluate 0 as false, and any other value as true.

Comparison functions

The HP 49G provides comparison functions that you can use in conjunction with the conditional and looping structures. You access them from the Programming Test menu. For example, to test A in relation to B, use the following:

`A==B`	Returns true if A equals B.
`A≠B`	Returns true if A does not equal B.
`A<B`	Returns true if A is less than B.
`A>B`	Returns true if A is greater than B.
`A≤B`	Returns true if A is less than or equal to B.
`A≥B`	Returns true if A is greater than or equal to B.
`SAME(A,B)`	Returns true if A is exactly the same object as B.

Conditional and looping structures

The following conditional and looping commands are available:

- **IF** *comparison* **THEN** *code* **END**

  If *comparison* evaluates to true, that is a non-zero value, runs *code*.
- **IF** *comparison* **THEN** *code-1* **ELSE** *code-2* **END**

  If *comparison* evaluates to true, runs *code-1*. If *comparison* evaluates to false, *code-2* is run.
- **CASE** *expression-1* **THEN** *code-1* **END**
  expression-2 **THEN** *code-2* **END**
  . . .
  expression-n **THEN** *code-n* **END**
  END

  Runs the code that corresponds to the first expression in the structure that evaluates to true.

- **START (***start*, *end***)** *code* **NEXT**

  Runs *code*, increments *start*. Repeats until *start* > *end*. The *code* is always run at least once.

- **START (***start*, *end***)** *code* **STEP** *(incr)*

  Runs *code*, increments *start* by the number specified by *incr*. (*incr* can be an expression.) Repeats until *start* > *end*. The *code* is always run at least once.

- **FOR (***var*, *start*, *end***)** *code* **NEXT**

  Runs *code*, sets *var* to *start.* Increments *var*, and repeats until *var* > *end*. This is similar to **START… NEXT** except that you can use *var* in your code.

- **FOR (***var*, *start*, *end***)** *code* **STEP** *(incr)*

  Runs *code*, increments *var* by the number specified by *incr*. (*incr* can be an expression.) Repeats until *start* > *end*. This is similar to **START … STEP** except that you can use *var* in your code.

- **DO** *code* **UNTIL** *comparison* **END**

  Runs *code*, then tests to see if *comparison* evaluates to true. Ends if it does. Repeats *code* if it does not. The *code* is always run at least once.

- **WHILE** *comparison* **REPEAT** *code* **END**

  Checks if *comparison* evaluates to true. Runs *code* if it does. Repeats until test returns false. This is similar to **DO** …**UNTIL** except that *code* is not run if *comparison* evaluates to false the first time.

Example

The following example processes a list of numeric values that is stored in a variable named MARKS. It performs the following:

- It determines the number of elements in the list.
- For each element in the list, the program compares the element to the pass value:
  a. If the element is greater than or equal to the pass value, inserts “Pass” after the value.
  b. If the element is less than the pass mark, inserts “Fail” after the value. Note that this converts the value to a string.
- It replaces the original value with the string.

```
« @ Local variable S is used
@ to store the step number.
@ Step from 1 to the size of the list.
FOR(S,1,SIZE(MARKS))
@ Extracts the element from the list
  GET(MARKS,S) → E
@ Compares it to the pass mark, amends and
@ replaces with the new value.
  « IF E≥50 THEN
    E+" Pass" ▶ E
  ELSE
    E+" Fail" ▶ E
  END ;
  REPL(MARKS,S,{E})▶MARKS
» ;
NEXT
»
```

Trapping errors

By default, a program halts when it encounters an error. If you want sections of your program to deal with errors rather than halt the program, you need to include the sections inside error trapping structures. You can then specify actions to take when your program encounters errors, rather than halting the program. The following error trapping structures are available.

- **IFERR** *code* **THEN** *error-code* **END**

  If the program encounters an error while it is running *code*, the remaining code is skipped and *error-code* is run. If no errors are encountered in *code*, *error-code* is not run.

- **IFERR** *code* **THEN** *error-code* **ELSE** *noerror-code* **END**

  If the program encounters an error while it is running *code*, the remaining code is skipped and *error-code* is run. If no errors are encountered in *code*, *noerror-code* is run.

Example

The following example creates the list of marks used in the previous example. If a non-numeric value is entered, the program prompts with an error message. The program performs the following:

- It sets up a loop to collect 20 values.
- It prompts for an input value.
- It tests the input to check if it is a number.
- If the generates an error, the error is trapped, and an error message is displayed to prompt for a numeric value.

```
«
@ Set numeric mode so that error trap works
SF(-3) ;
@ Create an empty list
{}▶MARKS ;
@ Set up a loop for 20 entries.
WHILE SIZE(MARKS)<20 REPEAT
@ Start error-checking routine.
IFERR INPUT("Enter a number","") → N
@ Attempt to convert the entry to a number.
@ This generates an error if non-numeric
« OBJ→(N)+1-1 → N
@ If no error, append the entry to the list.
« MARKS+N ▶ MARKS
»
»
THEN
@This appears if entry is non-numeric.
MSGBOX("INVALID ENTRY, TRY AGAIN")
END ;
END
»
```