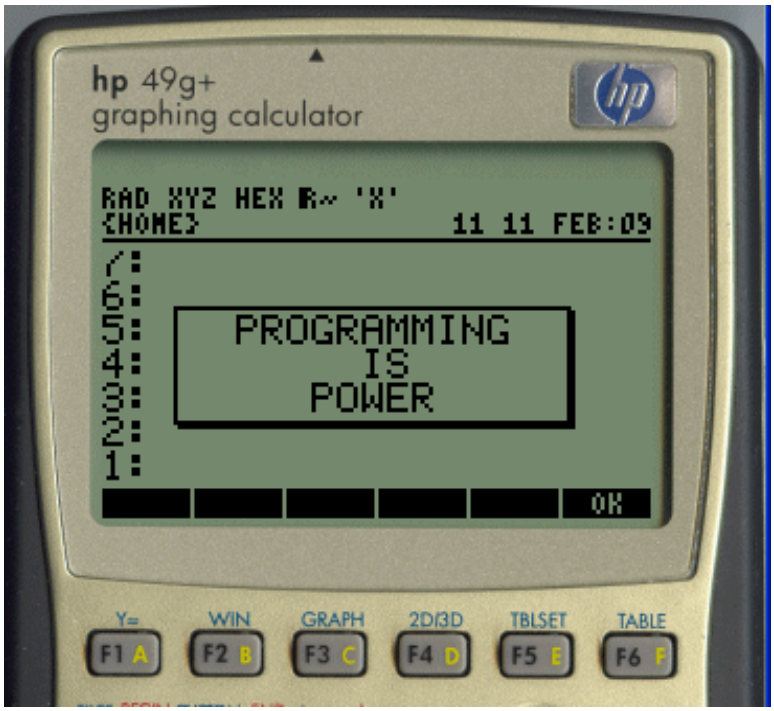
Return to Cover Page

An Introduction to Programming HP Graphing Calculators

Second Edition

by

Mervin E. Newton
Thiel College



To

Sara Franco Newton

Table of Contents

Preface

This introduction to programming HP graphics calculators assumes the reader is already familiar with the basic operation of the calculator; it deals only with programming techniques. The Web site HP 50g Calculator Tutorial

http://www.thiel.edu/MathProject/CalculatorLessons/Default.htm

provides an introduction to the basic operation to the three models mentioned below. These instructions are based on the HP 49G+, but much of what is included here applies to many other models. In particular, virtually everything here applies to the HP 48GII and the HP 50g, which are almost identical to the 49G+. The only differences are that the memory capacity of the 48 is much smaller and the shift keys on both are different colors. This introduction deals primarily with the programming techniques that will be needed by students in the Numerical Analysis course at Thiel College. There is much more to be learned by reading chapters 21 and 22 of *HP 49G+ Graphing Calculator User's Guide*. This publication comes on a CD with the calculator. This guide will be referred to as ***UG*** in what follows.

The HP 49G+ has three shift keys, the yellow ALPHA shift, the green left shift and the red right shift. They will be abbreviated **AS, LS**, and **RS** respectively. The four arrow keys; up arrow, down arrow, left arrow, and right arrow; will be abbreviated **UA, DA, LA**, and **RA** respectively. In some cases, especially with the calculator in RPN mode, a shift key must be held down while another key is pressed. In such cases the command will be written as **LS(hold)**. Finally, menu commands will be preceded by the appropriate soft key, **F1, F2, ..., F6**. For example, if the calculator is in algebraic mode, the following steps will put it into RPN mode:

MODE F2-CHOOS DA F6-OK F6-0K.

As another example, let us set the calculator to show soft menus. Press **MODE**, then **F1-FLAGS** then **UA** 7 times to highlight system flag 117. If this flag is not checked, press **F3-CHK**. The flag should now show "Soft MENU." Now press **F6-OK** twice. For all of the examples in this text it is assumed that the calculator is set for RPN and soft menus.

The author wishes to thank Sara Franco Newton, Cassandra Beck(ʻ09), Angela Crone(ʻ09), Klotilda Lazaj(ʻ04), Nicholas Lias(ʻ04), Kara McDowell(ʻ01), Amanda McKeehan(ʻ04), Andrew Murrin(post grad student), Anthony Ross(ʻ04), Michael Ryan(ʻ09), John Svirbly('06), Timothy Van Horn(ʻ09) Nicole Volchko(ʻ00), Sean Weaver(ʻ04), and Rebeccah Williams(ʻ04) for their help in proofreading this and previous editions of this work. Thanks also to Michelle Porada (Thiel class of 2000) for her help in preparing it for the WWW.

An interactive version of this text can be found on the WWW at

http://www.thiel.edu/mathproject/itphpc/

Section 1 - What is a Program?

It is suggested that before we start to write and save programs, we should create a directory into which to put them. To do this key **LS(hold) UPDIR** to get to the Home directory. Now key **LS FILES F6-OK NXT F3-NEW DA**. Type in your choice of name for the new directory, then **DA F3-CHK F6-OK**. Finally, press **CANCEL** (the ON button) to get out of the FILE dialog box. Now press **VAR** and press the soft menu key for the new directory you just created (it should be F1.) You are now in your new directory and ready to start saving new programs in it. For more about directories and saving variables see ***UG*** 2-32 to 2-59.

Basically, an HP49G+ program is an object that contains a listing of the steps to solve a particular type of problem. Let us look at a simple example. In a typical bowling league each member of each team bowls three games in a match. To find the average for a particular member for the match, you must add the individual's three scores and divide by 3. If the three scores were on the stack of your HP49G+, you would push the plus sign twice, enter a 3, and push the division key. Let us make this a program. On the HP49G+ the listing of a program must be enclosed in the symbols << >>, which is a right shifted plus sign. Our program then, would be

<< + + 3 ÷ >>

Type the above program into the command line by keying

RS <<>> + + 3 ÷

then press **ENTER** to put it on level 1 of the stack. Notice that as soon as you press **RS << >>**, you see the **PRG** annunciator in the upper right corner of the display, telling you that the calculator is in program mode. Also notice that division is shown as "/" in the program. For a program to be useful, it must be given a name and saved. Let us call it **BWL1**. Type **'BWL1'** into the command line and press **STO**. If you now press **VAR** you should see **BWL1** as the first item in your menu. To run the program put three scores on the stack and press the soft key for **BWL1**. Try it with the scores 172, 188, and 186: put the three numbers on stack levels 1, 2 and 3, then press the **BWL1** menu key. You should see 182 on level 1 of the stack. (NOTE: The last number in the previous example could, alternatively, have been left in the command line; as in most HP49G+ operations.)

The program above is short enough to fit in one line on the screen, but that will not always be the case. When entering longer programs it is helpful to use the carriage return, ↵, (right shift decimal point) to go to a new line while in program mode. This also serves as a carriage return in a string. Recall that in the HP language a string is any group of characters surrounded by quotation marks.

EXERCISE SET 1

1. Write a program called **TAX1** to find a six percent sales tax for the amount in level 1 of the stack. HINT: If you have the program set the number format to Fix 2 the tax will be shown rounded to the nearest penny. To do this include the sequence

2 LS PRG NXT F4-MODES F1-FMT F2-FIX

at the beginning of the program. All you will see in the program is "2 FIX."

2. Your sales force works 5 days per week. Each person's pay for the week is $500 plus 5% of sales over $1000. For example: a salesperson who sells $1200 in the week would earn $500 plus 5% of $200, which is $510. Write a program called **PAY** to take the five daily sales figures from the stack and compute the person's pay for the week. WARNING: Be sure you do not penalize a person who sells less than $1000 for the week. The **MAX** function (**LS MTH F5-REAL F5-MAX**) should be helpful here. Function **MAX** is explained on page 3-13 of ***UG.***

3. In Pennsylvania (as in many other states) the exits on the interstate highways are numbered by the nearest mile marker, and the speed limit is 65 MPH. Write a program called **TMR1** to take the exit number and the mile marker you just passed from the stack and compute the time it will take you to reach the exit assuming you are traveling at 65 MPH. NOTE: The **ABS** function (left shifted division key) should be useful to make sure you don't end up with a negative time. Function **→HMS** (**LS PRG NXT NXT F1-TIME NXT F3 −→HMS**), which is explained on page 25-3 of ***UG*** may also be helpful here. Since an answer to the nearest minute would normally be good enough for this type of problem, it would also make sense to set the display to FIX 2 for this program.

Section 2 - Stack Manipulation

When working on the HP49G+ in RPN mode we sometimes make use of the stack manipulation commands that are available in the **STACK** menu. These tend to be much more important in programming. With the calculator in RPN mode, the **STACK** menu can be found by pressing **TOOL F3-STACK** or **LS PRG F1-STACK**. There are 19 stack manipulation commands in this menu. These are listed and explained in the table below because they don't seem to be explained anywhere in ***UG***.

DUP	Duplicates the item on level 1 of the stack, puts it on level 1 of the stack and moves everything else up on the stack.
SWAP	Interchanges the items on levels 1 and 2 of the stack
DROP	Deletes the item on level 1 of the stack and drops everything else down one level.
OVER	Makes a copy of the item from level 2 of the stack, puts it on level 1, and moves everything else up one level.
ROT	Rotates the item in level 3 of the stack to level 1, the item from level 1 to level 2, and the item from level 2 to level 3.
UNROT	Reverses the process of ROT.
ROLL	With an integer *n* in the command line, ROLL works like ROT, but on the first *n* levels of the stack, so 3 ROLL has the same effect as ROT. If the command line in empty but there is an integer *n* on level 1 of the stack, ROLL takes *n* from the stack, drops the rest of the stack down one level, then performs the ROLL. Thus, 4 ROLL and 4 ENTER ROLL have the same effect.
ROLLD	Reverses ROLL
PICK	With an integer *n* in the command line, PICK works like OVER, but selects the item from level *n* of the stack, 2 PICK is the same as OVER. If the command line is empty and an integer *n* is on level 1 of the stack, PICK removes the integer from the stack, moves everything down one level, then performs the PICK, thus 4 PICK and 4 ENTER PICK have the same effect.
UNPICK	Is NOT quite the reverse of PICK. With an integer *n* in the command line, PICK removes the item in level 1 of the stack, moves everything in the stack down one level, then replaces the item in level *n* with the item that was originally removed from level 1. If the command line is empty and there is an integer *n* on level 1 of the stack, UNPICK removes the integer from the stack, moves everything in the stack down one level, then performs the UNPICK, thus 4 UNPICK and 4 ENTER UNPICK have the same effect.
PICK3	The same as 3 PICK
DEPTH	Counts the number of elements in the stack, puts the number on level 1 of the stack and moves everything else up one level.
DUP2	Duplicates the items in levels 1 and 2 of the stack and moves everything up 2 levels.

DUPN	With an integer *n* in the command line, DUPN duplicates the items in levels 1 through *n* and moves everything up *n* on the stack. If the command line is empty and there is an integer *n* on level 1 of the stack, DUPN takes the *n* from the stack, drops everything down one level, then executes the DUPN. Thus, 3 DUPN and 3 ENTER DUPN have the same effect.
DROP2	Drops the items on levels 1 and 2 of the stack and moves everything else down two levels.
DROPN	With an integer *n* in the command line, DROPN drops the items on levels 1 through *n* from the stack and moves everything else down *n* levels. If the command line is empty and there is an integer *n* on level 1 of the stack, DROPN removes the *n*, drops everything down one level, then executes the DROPN. Thus, 4 DROPN and 4 ENTER DROPN have the same effect.
DUPDUP	The same DUP DUP, that is, the same as pressing DUP twice.
NIP	Drops the item in level 2 of the stack and moves anything above level 2 down one level.
NDUPN	With an integer *n* in the command line, NDUPN puts *n* copies of the item on level 1 into levels 2 through *n* + 1, and *n* is put on level 1. Everything else on the stack is moved up *n* levels. If the command line is empty and there is an integer *n* on level 1, NDUPN removes *n*, drops everything on the stack down one level, then executes NDUPN. Thus, 3 NDUPN and 3 ENTER NDUPN have the same effect.

Suppose the bowling league of Section 1 computes a bowler's handicap for the next week as 80% of (200 minus this week's average). Let's write a program to take three scores from the stack and leave the total on level 3, the average truncated to an integer on level 2, and next week's handicap truncated to an integer on level 1.

```
<<  +  +  DUP 3  ÷  FLOOR 200
    OVER  -  0 MAX .8  ×  FLOOR >>
```

Type this program into your calculator and store it as **BWL2** then try it with several sets of data. With 172, 177 and 186, the output should be 535 on level 3, 178 on level 2, and 17 on level 1. Notice that the commands **0 MAX** are needed to make sure that a bowler whose average is over 200 will not have a negative handicap. The **FLOOR** function (**LS MTH F5-REAL NXT NXT F3-FLOOR**) is described on page 3-14 of ***UG***.

EXERCISE SET 2

1. Change Problem 3 of Exercise Set 1 so that the distance is left on level 2 of the stack and the time on level 1. Save this new version as **TMR2**.

2. Change Problem 1 of Exercise Set 1 so that the original amount is left on level 3 of the stack, the tax on level 2, and the total on level 1. Call this new version **TAX2**.

3. Write a program called **LIN1** to do linear interpolation. That is, given the two points (x_1, y_1), (x_2, y_2) and a number x between x_1 and x_2, find $y = y_1 + \frac{y_2 - y_1}{x_2 - x_1}(x - x_1)$. The program should begin with x_1 in level 5, y_1 in level 4, x_2 in level 3, y_2 in level 2, and x in level 1; and should end with y in level 1 and the rest of the stack empty. (First strive for a solution which works, but then try to do it in as few steps as possible.)

Section 3 - Editing and Debugging

It is an unfortunate fact of life that humans make mistakes (called bugs in this context) that cause the calculator to halt in mid program because it has been asked to do something impossible or that cause the calculator to give wrong answers. One of the easiest ways to find a bug in a program is to have the calculator execute the program one step at a time so we can see exactly what it is doing. Let's look again at the example **BWL2** from Section 2. Put three bowling scores on levels 4, 3 and 2, and **'BWL2'** on level 1. Now press **LS PRG NXT NXT F3-RUN F1-DBUG**. The program name will be taken off the stack and the annunciator **HLT** will appear at the top of the screen. Now press **F2-SST** once for each step of the program. Notice that with each press the step being executed is shown in the upper left corner of the screen and the result is shown on the stack. If you find an error and choose not to continue single stepping through the program, press the **F6-KILL**, to return to normal keyboard commands.

Once an error is found we need to edit our program. Suppose we want to edit **BWL2**. Enter **'BLW2'** on the stack and press **LS DA**. The program will now show in the command line and you can use the arrow keys to move around the program. A handy thing to remember, especially when you are editing a long program, is that **RS DA** gets you to the end of the program and **RS UA** gets you to the beginning of the program. Also, **RS LA** gets you to the beginning of a line and **RS RA** gets you to the end of a line. When the cursor is where the changes are needed, you can type in new things and use the backspace or delete keys to remove things. When the editor is entered it is in insert mode. The soft menu key **F6-INS** toggles between insert mode and overwrite mode. When the menu shows a small square after **INS**, insert mode is on.

While working in the command line, it is possible to bring up any of the menus that may

be useful in writing the program, such as **PRG, MTH**, etc. To return to the edit commands, press **TOOL.**

When all the necessary corrections have been made, press **ENTER** to make the changes permanent and return to normal calculator operation. If you decide you wish to exit without making the changes permanent, press the **CANCEL** key and the program will be left unaltered.

There are also several editing commands in the menu. Some of these are explained in Appendix L of ***UG***, but several are missing from the appendix and others don't do what the appendix says they do. On the first page of menu commands are **F1 – ← SKIP,** moves the cursor to the end of the previous word, **F2 – SKIP →** moves the cursor to the beginning of the next word, **F3 – ← DEL** deletes to the end of the previous word, and **F4 – DEL →** deletes to the beginning of the next word. **F5-DEL L** deletes an entire line, and **F6-INS** was discussed above.

On the second page of menus there are two subdirectories that are very useful. The first is **F1-SEARCH**. **F1-FIND** opens a dialog box with a field for the string to be searched for. After entering the target string, **F6-OK** finds the first occurrence of the string starting at the current cursor position and searching forward. If the string is not found before the end of the file, an error message is displayed. If the string is found, it is highlighted. To find the next occurrence of the string, press **F3-NEXT**. **F2-REPL** opens a dialog box with fields for the target string and for the replacement string. After entering the desired strings in these two fields, **F6-OK** will highlight the first occurrence of the target string (or give error message if it is not found). There are now three choices of how to make the replacement. **F4-R** makes the replacement and leaves the cursor at that point, **F5-R/N** will make the replacement and highlight the next occurrence of the target string, **F6-ALL** will make the replacement for all occurrences of the target string from the current location to the end of the file.

The other subdirectory of interest in the second page of the Edit directory is **F2-GOTO**. The one command in this subdirectory that is useful in long programs is **F1-GOTOL**. It will open a dialog box with a field for a line number. When a line number is entered and **F6-OK** is pressed, the cursor will move to the beginning of the requested line.

On the main keyboard there are also commands to do COPY/CUT/PASTE similar to what is available in most computer text editors. Place the cursor at the beginning of the text to be copied or moved and press **RS BEGIN**. Move the cursor to the end of the desired text and press **RS END**. Now press **RS COPY** or **RS CUT**, as desired, move the cursor to the point where the text is to be inserted and press **RS PASTE**.

In some cases (as in Problems 1 and 2 of Exercise Set 2) you may wish to create a new program by making changes to an existing program, but leave the original intact. To do this, press **RS** and press the menu key for the desired program to put it on the stack, then **DA** to put it on the command line. Make the necessary changes and press **ENTER**. The new version is now on the stack and can be saved under a new name, but the original is unaltered.

Section 4 - Variables, Input, Output

In all of our examples so far we have placed the necessary data on the stack before the program starts. This works but it can create problems for both the programmer and the user. This is best demonstrated by Problem 3 of Exercise Set 2; manipulating the stack with five values is somewhat of a challenge for the programmer, and the user must remember the exact order in which the data must be entered for the program to work correctly. It would also be helpful to the user if programs like **BWL2**, which have several results, had each output item labeled. In this section we will learn about the use of variables, which helps reduce the amount of stack manipulation that may be required by a program, and we will learn about some input and output techniques that will make the programs more user-friendly.

There are two types of variables available for programming, global and local. The variables you learned to use on pages 2-46 through 2-60 of ***UG*** are global variables. They function in programs just as they do in normal operation. Local variables are specific to a program and live only while the program is running. We will adopt the convention of using upper case letters for global variables and lower case letters for local variables. To create local variables you must first put the values on the stack, put in an arrow, **→,** (right shifted zero key), then the list of variable names, and finally a new set of << >> (like a program within the program). The local variables are now available to this inner program and can be used the same as global variables. We will see an example of this below.

The problem of forcing the user to memorize the order in which data must be entered can be solved with the **INPUT** command. This command allows the program to prompt the user for the data one item at a time. To use the command there must be a prompt (a string telling the user what to enter) on level 2 of the stack, "" on level one of the stack, then the **INPUT** command, which can be found at **LS PRG NXT F5-IN F4-INPUT**. When the command is executed the program pauses with the prompt displayed at the top of the screen and the cursor set to the command line ready for input. When the user presses **ENTER** the program resumes and places the input on level 1 of the stack as a string. Because the input is a string the most common command to follow the **INPUT** command is the **OBJ→** command, which can be found at **LS PRG F5-TYPE F1 - OBJ→.** We will see an example of this below.

We will discuss two methods of labeling output, tagged output and the message box. Tagged output is generally used at the end of a program. It leaves one or more output items on the stack, each with an explanatory tag. The message box is generally used to pause the program with intermediate results during program execution. After a message box comes up, the user must acknowledge it by pressing **F6-OK** for the program to continue.

To create a tagged output item the output object should be on stack level 2 and the desired explanatory tag on level 1 as a string. Then the **→TAG** command, which is found by **LS PRG F5-TYPE F5 –→TAG,** will combine them into a tag object on level 1. For the

message box, put a message in a string on level 1 and issue the **MSGBOX** command, which is found by **LS PRG NXT F6-OUT F6-MSGBO**. When the command is executed the string will be dropped from the stack. To create the message string, use the + to concatenate the parts. For example, suppose you wish to display the message "There are # apples in the bag.", where # is a number that is on level 1 of the stack. Use the following steps:

<< ... **"There are " SWAP + " apples in the bag." + MSGBOX** ... >>.

Notice that it was not necessary to convert the number to a string; when + operates on a string and a number it converts the number into a string then concatenates. If, however, you wish to concatenate two numbers, you must first explicitly convert at least one of them to a string with **LS PRG F5-TYPE F4 →STR**. It is also important to note that when + is used for concatenation it is not a commutative operation; the object in level 2 will appear in front of the object in level 1.

We will now consider two new versions of our bowling program; the first will demonstrate the use of global variables and the message box and the second will demonstrate the use of the **INPUT** command, local variables, and tagged output.

For the first example, which we call **BWL3**, we will make the following assumptions. During the play of the games we were using a score keeping program that left the scores of the three games in the calculator memory as **SCR1, SCR2,** and **SCR3**, and we want **BWL3** to leave the new handicap in memory as **HNCP** for use by the score keeping program next week. With this in mind, the coding for **BWL3** is

```
<<
    SCR1 SCR2 + SCR3 + "Series total = " OVER + MSGBOX
    3 / FLOOR "Average = " OVER + MSGBOX 200 SWAP - 0 MAX
    .8 * FLOOR "New handicap = " OVER + MSGBOX 'HNCP' STO
>>
```

Try single stepping through this program to see exactly how it works. Notice that you will have to create the global variable **SCR1, SCR2,** and **SCR3** before you can run the program. It is not necessary to create **HNCP** in advance, the calculator will create it if it does not exist, and will overwrite the existing version if it does exist.

The second example, which we will call **BWL4**, will use the **INPUT** command, local variables, and tagged output. Admittedly, the use of local variables in this case is not particularly helpful, but it does provide an example. Unlike **BWL3**, this program is intended to stand alone.

HINT: As you enter this program make use of the **COPY PASTE** feature to simplify entering the first three lines.

```
<<
    "GAME 1 SCORE" "" INPUT OBJ→
    "GAME 2 SCORE" "" INPUT OBJ→
    "GAME 3 SCORE" "" INPUT OBJ→
    0 0 0 --> s1 s2 s3 tot avrg hncp
    <<
        s1 s2 + s3 + 'tot' STO
        tot 3 / FLOOR 'avrg' STO
        200 avrg - 0 MAX .8 * FLOOR 'hncp' STO
        tot "TOTAL" →TAG
        avrg "AVERAGE" →TAG
        hncp "HANDICAP" →TAG
    >>
>>
```

Notice that to create the local variables **tot, avrg,** and **hncp** it was necessary to put an initial value of 0 on the stack for each of them. The initial value did not have to be zero, it could have been anything, but each must be initialized. Again, try single stepping through this program to see exactly how it works.

The HP49G+ has several other input and output features that will not be discussed in this text. Those interested in these more "exotic" I/O forms should consult pages 21-19 through 21-43 of ***UG***..

EXERCISE SET 4

Use the editing suggestions from Section 3 to make the required changes..

1. Change Problem 2 of Exercise Set 2 to add tags to the three lines of output. Save the new version as **TAX3**.

2. Change Problem 1 of Exercise Set 2 so that the desired exit number and the speed limit are obtained from global variables and the current mile marker is entered with an **INPUT** command. The distance and time should be reported by way of message boxes. The message for the time should read something like “Time = 2 hours and 12 minutes." There are two functions that should help make your output “pretty.” One is **LS MTH F5-REAL NXT NXT F1-RND** (see page 3-14 of ***UG***). The other is **LS CONVERT F4-REWRITE NXT NXT F1 – R→I** (see page 5-28 of ***UG***). Save this version as **TMR3**.

3. Change Problem 3 of Exercise Set 2 so that the values of x_1, y_1, x_2, y_2, and x are entered with **INPUT** commands and are assigned to local variables. The output, y, should be tagged. Save the new version as **LIN2**.

Section 5 - Branching

We will discuss two general types of branching, **IF** and **CASE**. The **IF** structure has the form:

IF *condition* **THEN**
  block 1
ELSE
  block 2
END

The *condition* is an expression that is either true or false and each of the *blocks* is a set of one or more instructions. If the *condition* is true then all the instructions in *block* 1 are executed, if the *condition* is false, then all the instructions in *block* 2 are executed. The **ELSE** *block* 2 is optional.

The **CASE** structure has the form

CASE
  condition 1 **THEN**
    block 1
  END
  condition 2 **THEN**
    block 2
  END
  .
  .
  .
  condition n **THEN**
    block n
  END
  block n + 1
END

The block of instructions corresponding to the first true condition encountered is the only block that will be executed. If none of the conditions are true, *block n* + 1 will be executed. This last block is optional.

Before we can continue with the **IF** and **CASE** structure, we need to understand how the HP49G+ works with boolean values and boolean operations, that is, with values that are either true or false. **LS PRG F4-TEST** brings up a menu with six relational functions (five of these functions are also on the keyboard). Each of the functions takes values from level 2 and level 1 and returns a 1 if the relation is true and 0 if the relation is false. For example, put 5 on level 2 and 8 on level 1. If you press **F3-<** or **RS <** you get a 1 and if you press **F4->** or **RS >** you get a 0. **F1-==** in this menu and **RS =** on the keyboard are NOT the same. In the HP language, = is used to assign a function to its name, == is used to test equality between two values. The == can be created in a program from the keyboard by **AS(hold) RS = RS =** with the alpha shift held for the four following keystrokes.

Press **NXT** to get to the next page of this menu and you find the commands **AND**, **OR**, **XOR**, and **NOT**. These four operations are the same on the calculator as they are in logic; **AND** is true if both conditions are true, **OR** is true if at least one condition is true, **XOR** is true if exactly one of the conditions is true and **NOT** is true if the condition is false. The first three of these are binary functions which act on levels 1 and 2 of the stack. **AND** returns 1 if both values are 1's and 0 otherwise. **OR** returns 1 if at least one of the values is a 1, and 0 otherwise. **XOR** returns 1 if exactly one of the values is 1 and the other is 0, and returns 0 otherwise. **NOT** toggles the value on level 1 between 0 and 1. It should be noted that these four logical functions and also the **THEN** command which we will discuss in the next paragraph, treat any nonzero real number as a 1. If, for example, you place .5 on level 2 of the stack and -3.7 on level 1 and press **AND** the response will be a 1.

We are now ready to return to the **IF** and **CASE** structures. The critical element here is the **THEN** statement. When the calculator comes to **THEN** it takes the value off of level 1 of the stack. If the value is true (that is, any real number not equal to zero), it executes the block of instruction immediately following, and at the end of that block goes to the very end of the **IF** or **CASE** structure. If the value is false (i.e., a zero) the program skips to the next section of the structure it is working through; in the case of the **IF** statement it goes to the block after the **ELSE** if there is one, or to the **END** of the **IF** if there is no **ELSE**; in a **CASE** structure it goes to the next condition, to *block n* + 1, or the **END** of the **CASE**, whichever comes first.

LS PRG F3-BRCH F1-IF gets one to the menu with the **IF** elements and **LS PRG F3-BRCH F2-CASE** to the menu with the **CASE** elements, but that's not the best way to get those structures into a program. From the **BRCH** menu **LS F1-IF** will put **IF ... THEN ... END** into the program and **RS F1-IF** will put **IF ... THEN ... ELSE ... END** into the program. **LS F2-CASE** will put **CASE ... THEN ... END ... END** into the program and each **RS F2-CASE** will put in an additional **THEN ... END**. These typing aids make the job easier and help eliminate the very common error of forgetting to put in an **END** statement.

We will write a program to calculate a person's pay for a week. The input will be the hourly rate and the hours worked. If the hours worked is greater than 40, the person is to receive an extra half pay for the hours over 40.

```
<<
    "Hours worked" "" INPUT OBJ→
    "Pay rate" "" INPUT OBJ→
    → h pr
        <<
            h pr *
            IF h 40 > THEN
                h 40 - pr 2 / *  +
            END
        >>
    "Gross pay" →TAG
>>
```

Enter this program and save it as **GPAY**. Single step through the program three times with the hours worked less than, equal to, and greater than 40.

We will now write a new version of **BWL3** that will use an **IF** statement to overcome the problem of a negative handicap if the bowler's average is greater than 200.

```
<<
    SCR1 SCR2 + SCR3 + "Series total = " OVER + MSGBOX
    3 / FLOOR "Average = " OVER + MSGBOX
    IF DUP 200 < THEN
        200 SWAP - .8 * FLOOR
    ELSE
        DROP 0
    END
    "New handicap = " OVER + MSGBOX 'HNCP' STO
>>
```

Save this program as **BWL5** and single step through it a couple of times, once with the average less that 200 and once with it over 200. **BWL3** is actually a more efficient program than this one, but this gives us a good example of how the **IF** statement works, and it is easier to read this program and understand what it is doing.

For an example of a **CASE** structure we will write a program to solve the quadratic equation $ax^2 + bx + c = 0$. We will assume we are only interested in real solutions, so we will write the program to output a message about the type of roots, but only give the values in the case a double root or of two distinct real roots.

```
<<
    "Enter the leading ↵ coefficient" "" INPUT OBJ→
    "Enter the coefficient ↵ of the linear term" "" INPUT OBJ→
    "Enter the constant ↵ term" "" INPUT OBJ→
    0 → a b c d
      <<
        b SQ 4 a * c * - 'd' STO
        CASE
          d 0 >  THEN
                    "The roots are " b NEG d √x + 2 a * /
                    DUP
                    IF FP 0 == THEN
                        R→I
                    END
                    + " and " + b NEG d √x - 2 a * /
                    DUP UNROT + SWAP
                    IF FP 0 ≠ THEN
                        "." +
                    END
                END
          d 0 == THEN
                    "There is a double root at " b NEG 2 a * /
                    DUP UNROT + SWAP
                    IF FP 0 ≠ THEN
                        "." +
                    END
                END
          "The roots are complex."
        END
      >>
    MSGBOX
>>
```

The **NEG** function to change the sign of the element in level 1 can be entered by pressing the +/- key. The "extra" **IF** statements that include the function **FP** are there to make the output "pretty." **FP** can be found at **LS MTH F5-REAL NXT F6-FP**. It is explained on page 3-14 of ***UG***. Save this program as **QDEQ** and single step through it with different values of *a*, *b*, and *c*

so that you can see all three paths of the **CASE** in action. Be sure to try examples with both integer and non-integer roots to see why the "extra" **IF** statements and the use of **FP** and **R→I** were necessary.

Tuition at Podunk University is $25000 per year. A student who is the child of a PU employee or the child of a clergy gets a 10% discount. If the student is the child of both a PU employee and a clergy, the discount is 15%. We will write a program that will ask if the student fits either of those conditions, then compute the bill.

```
<<
  "Child of employee? ↵ Enter Y or N" { α "" } INPUT
  "Child of clergy? ↵ Enter Y or N" { α "" } INPUT
  1 → e c m
  <<
    CASE
      e "Y" == c "Y" == AND THEN .85 'm' STO END
      e "Y" == c "Y" == OR THEN .9 'm' STO END
    END
    25000 m *
  >>
  "Tuition" →TAG
>>
```

The coding **{ α "" }** puts the calculator into alphabetic mode before the **INPUT** is executed, so the user need only press the appropriate letter key, without pressing **AS** first. The question mark, **?,** can be entered by **AS RS 3**. Save this program as **TUIT** and single step through it with various choices of input to see how the **AND** and **OR** work.

EXERCISE SET 5

1. Add the coding necessary to your solution of Problem 2 of Exercise Set 4 so that if the time to the desired exit is more than four hours a message box will warn the user of the need to plan an intermediate rest stop. Call this program **TMR4**.

2. Write a program to check a customer's credit availability. The user should be prompted to enter the customer's outstanding balance, current purchase, and credit limit. If the balance plus the current purchase is less than or equal to the credit limit a message box will say the purchase is approved, otherwise it will disapprove the purchase. Call this program **CRD**

3. Change the example **GPAY** above to compute withholdings and net pay. If the employee

earned no more than $50, there is no withholding. If the earnings are more than $50 but no more than $300, withhold 10% of the excess over $50. If the earnings are over $300, but no more than $500, withhold $25 plus 15% of the excess over $300. If the earnings are over $500, withhold $55 plus 20% of the excess over $500. There should be three lines of tagged output giving the gross wages, the withholding, and the net wages. Call this program **NPAY1**

4. Change Problem 3 of Exercise Set 4 so that if x is not between x_1 and x_2 inclusive a message box will give the user a warning that the program is extrapolating. Save this program as **LIN3**.

Section 6 - Looping

One of the great powers of computer and calculator programs comes from the ability to repeat a sequence of commands as many times as may be necessary. The HP49G+ provides five different looping structures, only three of which will be discussed here. Those interested in the other looping forms should consult pages 21-53 through 21-63 of ***UG***..

The first looping structure we will explore is the **WHILE** structure. It has the form

WHILE *condition* **REPEAT**
    block
END

The *condition* here is the same as in Section 5 and the **REPEAT** command works very much like the **THEN** command discussed in Section 5. When the program reaches **REPEAT** the object is removed from level 1 of the stack, if it is true (i.e. any nonzero real number) the *block* of instructions is executed then the program returns to the *condition* and tests it again. Notice that something must happen to eventually make the *condition* false, or the program will loop forever. When the *condition* is false (i.e., the object on level 1 is a zero), the program jumps to the first command after the **END**. All the elements for the **WHILE** structure can be found in **LS PRG F3-BRCH F6-WHILE**, but as with **IF**, **LS F6-WHILE** from the **BRCH** menu will produce the whole **WHILE...REPEAT...END** structure.

As an example we will write a program that is a variation on our earlier program to compute our bowling average. In this case we will assume that we are simply out for an evening of bowling with friends and don't know how many games we will bowl. We will have the calculator prompt the user for the next score until a negative score is entered, which will be a signal (called a sentinel) that there are no more scores.

```
<<
    0 0
    "Enter first score." "" INPUT OBJ→
```

```
    WHILE DUP 0 ≥ REPEAT
        + SWAP 1 + SWAP
        "Enter next score ↵ or -1 to quit." "" INPUT OBJ→
    END
    DROP
    IF SWAP DUP 0 > THEN
        / "Average" →TAG
    ELSE
        "No scores to average." MSGBOX DROP2
    END
>>
```

Save this program as **BOL1** and single step through it at least twice, once with several scores before the -1 is entered and once with -1 entered as the first score. See how it all works.

The second looping procedure we will consider is the **FOR** structure, which has the form

first last **FOR** *index*
    block
NEXT

In this structure *first* is an integer that represents the first value of the loop index, *last* is an integer that represents the last value of the loop index, and *index* **is** the loop index and is a local variable that lives only while the loop is in effect. **WARNING:** Because $i = \sqrt{-1}$ to the calculator, using *i* as the loop index can sometimes cause strange behavior. Although it is legal, it is better to choose some other letter for *index*. When the **FOR** structure is encountered *first* and *last* are removed from levels 2 and 1 of the stack respectively, the value *first* is assigned to *index* and the *block* is executed. When **NEXT** is encountered *index* is increased by 1; if it is still less than or equal to *last*, the *block* is executed again; if *index* is greater than *last*, the program jumps to the next command after **NEXT**. This continues until the loop is satisfied, that is, until *index* becomes greater than *last*. As soon as the loop is exited, the *index* is no longer available. The **FOR** structure can be found at **LS PRG F3-BRCH F4-FOR**. As in the previous structures we have studied, **LS FOR** from the **BRCH** menu types in the complete structure.

As an example, we shall rewrite **BOL1** to prompt the user for the number of games played, then ask for exactly that number of scores.

```
<<
    "How many games?" "" INPUT OBJ→
    IF DUP 0 > THEN
```

```
        → n
          <<
              0 1 n FOR k
                    "Enter score " k + "" INPUT OBJ→ +
              NEXT
              n / "Average" →TAG
          >>
    ELSE
          "No scores to average." MSGBOX DROP
    END
 >>
```

Save this program as **BOL2** and single step through it at least twice, once with a positive first input and once with zero as the first input. It won't show on this program because it is protected by the **IF** statement, but if *last* starts smaller than *first* the **FOR** loop will still be executed once.

A minor variation of the **FOR** loop structure is the **START** loop structure:

first last **START**
    block
NEXT

This is used to loop a fixed number of times when the value of the index is not needed. We could, for example have written the previous program as

```
<<
  "How many games?" "" INPUT OBJ→
  IF DUP 0 > THEN
    DUP 0 SWAP 1 SWAP START
      "Enter score." "" INPUT OBJ→ +
    NEXT
    SWAP / "Average" →TAG
  ELSE
    "No scores to average." MSGBOX DROP
  END
>>
```

This way, however, the prompt is not quite as user friendly. This version also demonstrates that it was not necessary to use the local variable *n*. Save this program as **BOL3** and try it with a

positive for the number of games and with 0 for the number of games. The **START...NEXT** structure can also be created from the **BRCH** menu with **LS F3-START**.

EXERCISE SET 6

1. Change Problem 1 of Exercise Set 4 to prompt the user to enter the price of each item the customer is purchasing until a zero is entered. (We are assuming that nothing costs $0, so it can serve as a sentinel.) It should then display the subtotal, the tax, and the total as tagged output. Call this version **TAX4.**

2. Professor Dinklesmith gives 10 quizzes during the semester and the final grade is the average of the 10 scores. Each quiz is worth 100 points. An average of 90 or better is an A, 80 or more but less than 90 is a B, 70 or more but less than 80 is a C, 60 or more but less than 70 is a D, and less than 60 is an F. Write a program that will prompt the user for the ten scores then output the average with the appropriate letter grade as a tag. For example, if the average is 87.3, the output should be **B:87.3**. Call this program **GRD**.

Section 7 - Flags

The HP49G+ has 256 user flags (numbered 1 to 256) that can be thought of as built in boolean variables. **LS PRG F4-TEST NXT NXT** will take you to a menu with six flag commands; **SF** = set flag, **CF** = clear flag, **FS?** = is flag set?, **FC?** = is flag clear?, **FS?C** = is flag set and clear it, and **FC?C** = is flag clear and clear it. To use any of the commands a flag number must be on level 1 of the stack. When the command is executed, the number is removed from the stack, and in the case of the last four commands, the appropriate response, 1 for yes or true and 0 for no or false, is placed on level 1. For more details on these commands see Chapter 24 of ***UG***.

Suppose our bowling league is for families with children and the kids get a little extra handicap depending on their ages: those at least 10 but less than 13 get an extra 5 pins, those at least 7 but less than 10 get an extra 10 pins, and those less than 7 get an extra 15 pins. We will rewrite **BWL4** to incorporate these new rules and call it **BWL6**. Run it with different choices for the ages of the bowler to see how it works.

```
<<
    "Is the bowler less ↵ than 13? Y/N" { α "" } INPUT
    IF "Y" == THEN
        "Enter age group: ↵ 10 ≤ age < 13 = 1
        ↵ 7 ≤ age < 10 = 2 ↵ age < 7 = 3."
        "" INPUT OBJ→ SF
```

```
    END

    "GAME 1 SCORE" "" INPUT OBJ→
    "GAME 2 SCORE" "" INPUT OBJ→
    "GAME 3 SCORE" "" INPUT OBJ→
    0 0 0 → s1 s2 s3 tot avrg hncp
    <<
        s1 s2 + s3 + 'tot' STO
        tot 3 / FLOOR 'avrg' STO
        200 avrg - 0 MAX .8 * FLOOR
        CASE
            1 FS?C THEN 5 + END
            2 FS?C THEN 10 + END
            3 FS?C THEN 15 + END
        END
        'hncp' STO
        tot "TOTAL" →TAG
        avrg "AVERAGE" →TAG
        hncp "HANDICAP" →TAG
    >>
>>
```

Notice that the program will leave no flags set when it ends. It is wise to make sure that any program that uses flags leaves them all clear at the end. If there are several flags to clear at once, their numbers can be put in a list on level 1 of the stack, the **CF** will clear them all.

EXERCISE SET 7

1. Change Problem 3 of Exercise Set 5 as follows: If the employee is over 62 or disabled, the withholdings are reduced by 10% of the initial calculation; if the employee is both over 62 and disabled, the withholdings are reduced by 15% of the initial calculation. Be sure to make appropriate use of flags. Save this program as **NPAY2**.

Section 8 - Arrays

It is assumed that the reader is familiar with the use of arrays and their associated commands on the keyboard and various menus. An array may be a vector or a matrix. These are discussed in chapters 9, 10 and 11 of ***UG***. Besides the **GET** and **PUT** commands to access specific elements of an array, there is an algebraic way of accessing them that is frequently much

more convenient in programs. In what follows let **v** be a vector and **M** be a matrix. Then '**v**(k) + 7' **EVAL** will get the kth element of **v**, add 7 to it, and leave the result on the stack. The sequence 12 '**M**(2,5)' **STO** will replace the element in the 2nd row and 5th column of **M** with the number 12. We will also be making use of several memory arithmetic commands in what follows. These commands are not described anywhere in ***UG*** so they are explained below. These commands can be found in **LS PRG F2-MEM F6-ARITH**.

The first four are **STO+**, **STO-**, **STOx**, and **STO/**. They require the name of a variable to be on either level 1 or level 2 of the stack and a constant on the other of level 1 or level 2. When one of these four commands is executed the contents of the variable replaces the variable name on the stack, the requested arithmetic is performed (if the values are compatible), and the result is removed from the stack and stored in the named variable. For example, suppose the vector [1, 3] is on level 2 of the stack and the variable 'A', which contains a 4, is on level 1. After executing **STOx**, the variable A will contain the vector [4, 12]. These four commands work with any combination of real numbers, complex numbers, and arrays so long as the requested arithmetic is defined. **STO+** also works for concatenation of lists and strings.

The fifth and sixth commands are **INCR** (increment) and **DECR** (decrement). These commands require the name of a variable on level 1 of the stack. For most applications the variable will contain an integer. Executing one of these commands will remove the name from the stack, increment or decrement the variable by 1 and leave the new value on level 1 of the stack.

On the second page of this menu are three more commands, **SINV** (inverse), **SNEG** (negative), and **SCONJ** (conjugate). They require the name of a variable on level 1 of the stack. Executing one of the commands will remove the name from the stack and take the inverse, negative, or conjugate of the contents of the variable.

Suppose that **A** is a vector of unknown length that contains integers and that has been stored in memory. We want to find the average of the elements of **A** to one decimal place and then determine if one of the elements of **A** is equal to that average. The **SIZE** command can be found at **LS MTH F2-MATRX F1-MAKE F6-SIZE** and **GET** is **LS MTH F2-MATRX F1-MAKE NXT F1-GET**.

```
<<
    A SIZE 1 GET 1 0 0 → n k sum av
      <<
          1 n FOR j
                'sum' 'A(j)' EVAL STO+
          NEXT
          sum n / 'av' STO 1
          WHILE n ≤ 1 FC? AND REPEAT
                IF 'A(k)==av' THEN
                      1 SF k
```

```
            ELSE
                'k' INCR
            END
        END
        "Average = " 1 FIX av + ". ↲  It is " + STD
        IF 1 FS?C THEN
            "The " + k R->I +
            CASE
                k 1 == THEN "st" END
                k 2 == THEN "nd" END
                k 3 == THEN "rd" END
                "th"
            END
            + " ↲  element of A."
        ELSE
            "NOT IN A."
        END
        + MSGBOX
    >>
>>
```

The command **STD**, found in **LS PRG NXT F4-MODES F1-FMT F1-STD**, puts the display back to standard. Save this program as **FDAV** and single step through it with several different vectors **A** to see how it works. Be sure to choose values that will test all the paths through the **IF** and **CASE** structures.

EXERCISE SET 8

1. With a vector **A** of unknown length as in the example **FDAV** above, write a program, **FDSM**, to go through the vector to find the smallest value and to see if there is more than one element with that smallest value. The output should include the smallest value, the index of the first occurrence of the smallest value, and a statement about whether or not that value is unique.

Section 9 - Procedures

It is possible for one program to call another program. The program that is called from another is referred to as a procedure. There are two main reasons for using procedures.

The first reason is to avoid rewriting the same code when the same thing must be done at several places in a program. For example, suppose you need to find the median of several sets of

numbers. The process of finding the median is the same for each set, only the numbers change. It would therefor be more efficient to write a procedure to find the median of an arbitrary set of numbers and calling it each time it is needed than to include the code for the complete process at each point.

Suppose a city is divided into quadrants. We have random samples of the cost (in thousands) of homes in each of the quadrants. The data for the north east quadrant is stored in a list called NE, for the north west quadrant in a list called NW, for the south west quadrant in a list called SW, and for the south east quadrant in a list called SE. (Recall that in HP terminology a "list" is a set of objects contained in braces, {}. Any type of objects can be contained in a list.) We wish to find the median cost of homes in each of the quadrants and in the city as a whole. The program **MEDN** below is a procedure to find the median of the numbers in a list on level 1 of the stack, and the program **FMED** calls that procedure 5 times to find the median of each quadrant and for the city as a whole. The program **SM** is a procedure to display each median. Enter the following three programs into your calculator.

```
<<
   NE MEDN "NE" SM NW MEDN "NW" SM SW MEDN "SW" SM SE MEDN
   "SE" SM NE NW + SW + SE + MEDN "CITY" SM
>>
```

Save the above program as **FMED**.

NOTE: When two lists are "added" with +, they are concatenated into one list. The + sign can also be used to add an object to a list. If the list is on level 2 and an object is on level 1, the object will be added to the end of the list. If the object is on level 2 and the list on level 1, the object is added to the beginning of the list. See Chapter 8 of ***UG*** for more information about list manipulation commands.

```
<<
    DUP SIZE → a n
   << a SORT 'a' STO
      IF n 2 MOD THEN
         a n 2 / CEIL GET
      ELSE
         n 2 / DUP 1 + a SWAP GET SWAP a SWAP GET + 2 /
      END
   >>
>>
```

The **CEIL** function, (**LS MTH F5-REAL NXT NXT F4-CEIL)**, is described on page 3-14 of

UG. Save the above program as **MEDN**.

<< "Median for ↵ " SWAP + " = " + SWAP + MSGBOX >>

Save the above program as **SM**.

Now create the four lists of data shown below and save them with the name indicated. These files, and the file needed in Section 10, can be downloaded from

http://www.thiel.edu/mathproject/Itphpc/default.htm

onto your computer then transferred to the calculator if you have the HP connectivity kit installed on your computer. The connectivity kit is on the CD that comes with the calculator or can be downloaded from the HP Web site at

http://www.hp.com/calculators/graphing/index.html

click on "Support" then follow the appropriate links.

NE = { 89 105 97 120 101 94 142 121 110 98 109 132 126 114 91 126 173 }

NW = { 234 159 197 348 288 175 268 368 312 }

SW = { 129 85 101 114 104 125 103 178 161 155 179 150 103 161 145 127 158
102 178 173 106 }

SE = { 105 131 131 120 94 90 142 138 122 87 83 136 109 129 98 91 121 141 128)

Now execute **FMED** and you should see the following results in message boxes:

"MEDIAN FOR NE = 110"
"MEDIAN FOR NW = 268"
"MEDIAN FOR SW = 129"
"MEDIAN FOR SE = 121"
"MEDIAN FOR CITY = 126"

NOTE: The sample sizes for the quadrants must be proportional to the number of homes in each quadrant for the CITY median to be valid.

If you do not get the correct answers, use the techniques discussed in Section 3 to try to correct the error. It is usually best to use the bottom up testing technique. That is, it is best to test

the lowest level procedures before the calling programs. To test **SM**, put a number on level 2 of the stack and a name in quotes on level one. Press the **SM** button and see if you get the right output. Next put a list of numbers on level 1 of the stack and press **MEDN** to see if the correct median is being computed. Be sure to test with both an even and with an odd number of data points in the list. Finally, check **FMED**. If you are using the single step procedure described in Section 3, note that in the **RUN** directory there is a command called **SST** and one called **SST↓**. If you use **SST** and you come to a procedure written by the user, such as a call to **MEDN** from **FMED** the procedure will simply be executed in one step. If you use **SST↓** the calculator will continue to single step through the called procedure. If one has done good bottom up testing, **SST↓** is almost never needed.

Notice that the procedures **MEDN** and **SM** are very different in nature. **MEDN** is a general purpose procedure that can easily be used as a stand alone program or as a procedure for another program. One can manually put a list of numbers on the stack and press the **MEDN** command or, as in the example above, have a program place a list on the stack and call **MEDN**. In either case, **MEDN** will find the median and leave it on the stack ready to be used as needed.

On the other hand, **SM** is a special purpose procedure intended specifically for **FMED** to display its results. It could, of course be used by other programs, but it was written with this particular program in mind. Clearly the coding for **SM** could have been included in **MEDN**, which would have made **FMED** shorter since it would have required only one procedure call for each data set, but it would have negated the generality of **MEDN**. This is a tradeoff that the intelligent programmer must always keep in mind. In this case it seemed more reasonable to keep **MEDN** general since it could have many other applications with the **SM** coding left out.

The other reason for using procedures is to make long programs easier to write, understand, and test. In the classical top down design approach to programming, a large problem is broken down into smaller parts called modules. Each module is then broken down into smaller modules until each module in a relatively simple problem. Then each module is coded as a special purpose procedure. For example, a payroll program may be broken down into modules to compute the gross pay, compute deductions, update year to date totals, and print a check. The gross pay module may have separate modules to compute regular pay and overtime pay. The deductions module could contain separate modules to compute federal taxes, state taxes, local taxes, insurances, 401K's, etc. We will not code this example as it would take too much time and space, but if you have a long program to write, this approach is highly recommended. The logic of the program is much easier to follow and the bottom up testing technique discussed above makes it much easier to find errors.

EXERCISE SET 9

1. Given two sides *a* and *b* and the included angle *C* of a triangle, the third side, *c*, can be found with the law of cosines $c = \sqrt{a^2 + b^2 - 2ab\cos(C)}$. Given the three sides *a, b, c* of a triangle, the area can be found by Heron's formula $area = \sqrt{s(s-a)(s-b)(s-c)}$, where $s = (a + b + c)/2$. Given four sides and one angle of a convex quadrilateral, the area can be found by drawing the diagonal between the two vertices that do not include the given angle. Use the law of cosines to find the length of the diagonal, then use Heron's formula to find the areas of the two triangles that are formed and add these areas. Write general purpose procedures for the law of cosines, **LCOS,** and for Heron's formula, **HERN**. Write a program **FDAR** that uses those two procedures to find the area of a convex quadrilateral given the four sides and one angle. FDAR should prompt the user for the input and the output should be tagged.

Section 10 - Recursion

It is possible for an HP program to call itself. This is called recursion. There are many cases in mathematics and computer science where recursion is an effective tool. Three classical examples are given below. The first two demonstrate effective uses of recursion, the third is an example of where recursion should NOT be used.

The first example is *n* factorial. This can be defined recursively as

$$0! = 1$$
$$n! = n(n\text{-}1)! \text{ for } n = 1, 2, 3, \ldots$$

This can be coded recursively as follows:

```
<<
    R→I
    IF DUP 0 == THEN
       DROP 1 R→I
    ELSE
       DUP 1 R→I - NFCT *
    END
>>
```

Key this into your calculator and save it as **NFCT**. Try it with 0 and 6. You should get 1 and 720 respectively. This is better than the built in factorial function (**LS MTH NXT F1-PROB**

F3-!) because it always gives and integer answer. The built in function converts to scientific notation at 15!, but it does give exact answers up to 17! because the missing digits are all zeros. At 18!, however, this program gives the exact value of 6402373705728000 while the built in function gives 6.40237370573E15. The use of **R→I** in sample program **QDEQ** was simply part of the output beautification process, but in this program it is necessary to keep the calculator working in integer mode to get the extra accuracy.

The second example is a binary search, which is a very fast way for finding an element of an ordered list. The program is written to "guess" that the middle element is the one being sought. If it is, the task is finished. If it is not, then the element being sought is to the left or to the right of the middle, so we recursively search the left half or the right half of the list, as the case may be. Suppose, for example, that we have the following list of records in the calculator where each record is a list consisting of a record number and a name. This file can be downloaded from the Web to your computer at http://www.thiel.edu/mathproject/Itphpc/default.htm then transfered to your calculator if you have the calculator connectivity kit installed on your computer.

NLST = { { 103 "BOB BUBBLES" } { 107 "DOLLY DULL" } { 123 "ANNY ANCHOR" } { 130 "QUINCY QUICK" } { 141 "LARRY LUCKY" } { 173 "HELEN HAPPY"} { 211 "ULMA UPSY" } { 229 "VIOLET VOCAL" } { 256 "OLGA OLF" } { 277 XAVIOR XANY" } { 282 "MARK MUMPS" } { 298 "WALTER WACKY" } { 317 "FRED FREEK" } { 333 "YOLANDA YELLER" } { 361 "NELL NURDY" } { 375 "TOM TALL" } { 380 "KATHY KRUNCH" } { 381 "PAT PICKLE" } { 382 "ZELDA ZILCH" } { 383 "JIM JELLY" } { 384 "SANDY SLIM" } { 385 "ED ELFY" } { 409 "CHUCK CHUNKY" } { 419 "RICH RAGS" } { 423 "IRMA IRKY" } { 431 "GUSS GUSHY" } }.

The objective is to write a program that will prompt the user for a record number and return the name of the corresponding person, or "NOT FOUND" if the number is not in the list. The solution is given below. The first program is a recursive binary search that takes the list, the first element to be searched, the last element to be searched, and the target number from the stack and returns the location of the required record, or zero if the record is not found. The second program prompts for the record number, sets up the stack to start the recursive process, and calls the search program. If the record was found it accesses the record and outputs the name, if the record was not found it outputs "NOT FOUND" to a message box.

```
<<
  → s a b t
  <<
    IF a b > THEN
      0
    ELSE
```

```
        a b + 2 / CEIL → c
        <<
          s c GET 1 GET
          IF DUP t == THEN
            DROP c
          ELSE
            IF t > THEN
              s a c 1 -
            ELSE
              s c 1 + b
            END
            t BSCH
          END
        >>
      END
    >>
>>
```

Save the above program as **BSCH**.

```
<<
  NLST DUP 1 SWAP SIZE "Enter record number" "" INPUT OBJ→
  BSCH
  IF DUP 0 == THEN
    DROP "NOT FOUND"
  ELSE
    NLST SWAP GET 2 GET
  END
  MSGBOX
>>
```

Save the above program as **FIND**. Enter the data set and try the program by pressing **FIND**. Study **BSCH** and make sure you understand how it works. Notice that **BSCH** will work on any list of records so long as the record number is the first element in the record. **FIND**, however, is specific to this particular list of records.

Another application of **BSCH** is to delete an element from the list. The following program prompts the user for a record number, sets up the stack for the recursive process and calls the search program. If the record was found, it is deleted from the list **NLST**. If the record is not found, "NOT FOUND" is output to the stack.

```
<<
   NLST DUP 1 SWAP SIZE "Enter record number" "" INPUT OBJ→
   BSCH
   IF DUP 0 == THEN
      DROP "NOT FOUND"
   ELSE
      → n
      <<
         CASE
            n 1 = THEN
                     NLST TAIL
                  END
            NLST SIZE n == THEN
                              NLST REVLIST TAIL REVLIST
                           END
            NLST 1 n 1 - SUB NLST n 1 + NLST SIZE SUB +
         END
         'NLST' STO "RECORD DELETED"
      >>
   END
   MSGBOX
>>
```

Save the above program as **DELT.** Save **NLST** with another name so you can restore it to the original after trying **DELT**, Be sure you understand how it all works.

The last example is using recursion to find the Fibonacci numbers. These are defined by

$$F_0 = 0,\ F_1 = 1, \text{ and } F_n = F_{n-1} + F_{n-2} \text{ for } n = 2, 3, 4, \ldots$$

This is easily coded as

```
<<
   → n
   <<
      IF n 1 ≤ THEN
         n
      ELSE
         'FIBR(n-1)' EVAL n 2 - FIBR +
      END
```

```
    >>
>>
```

Enter the above program and save it as **FIBR**. Notice that we have used two different methods for the recursive calls, algebraic: 'FIBR(n-1)' EVAL, and using RPN logic: n 2 - FIBR. There is no particular advantage of one method over the other, it was simply our intent to demonstrate both. Use the program to evaluate $F_6 = 8$.

Any program that can be written recursively can also be written as a loop. Below is a version of the Fibonacci program written as a loop.

```
<<
   → n
   <<
      IF n 1 ≤ THEN
         n
      ELSE
         0 1 2 n
         START
            SWAP OVER +
         NEXT
         SWAP DROP
      END
   >>
>>
```

Enter the above program and save it as **FIBL**. It is not nearly as elegant or easy to follow as the recursive version, but try finding F_{10} with both programs and you will easily see the advantage of the loop version. When recursion applies, it usually makes programming much easier, but as this example shows, it must be used with caution.

EXERCISE SET 10

1. One source of recursive formulas in mathematics is in the infinite series solutions to differential equations. A typical case for a second order differential equation has the form

$$a_0 = 1,\ a_1 = 1,\ a_j = f(j)a_{j-2} \text{ for } j = 2, 3, 4, \ldots$$

where $f(j)$ is some function of j. The solution of the differential equation is then

$$x(t) = b_0 \sum_{k=0}^{\infty} a_{2k} t^{2k} + b_1 \sum_{k=0}^{\infty} a_{2k+1} t^{2k+1}$$

where b_0 and b_1 are given by the initial conditions.

(a) Write a recursive procedure called FNDA that will evaluate the *a*'s given by

$$a_0 = 1,\ a_1 = 1,\ a_j = -\frac{j+2}{j(j-1)} a_{j-2},\ \text{for } j = 2, 3, 4, \ldots$$

(b) Write a program called FNDC that will prompt for the initial conditions b_0 and b_1 and output the values of the coefficients

$$b_0 a_0, b_1 a_1, b_0 a_2, b_1 a_3, \ldots, b_0 a_8, b_1 a_9$$

through the coefficient of t^9. The output should be tagged.

2. Alter **BSCH** so that given a new record number it returns the location where the new record should be inserted to maintain the list in order or a zero if the record number already exists. Call the new version **SRHB.** Write a program that prompts the user for the number and name of a new record. It should use **SRHB** to find where the new record should be inserted or if the record number already exists. It should then insert the new record into NLST or give an error message if the record number already exists. Call this program **NSRT.** Put **NLST** and all the programs associated with it into a directory and create a custom menu for that directory with **FIND, NSRT,** and **DELT**. See page 20-2 of ***UG*** for instructions on how to create a custom menu.

INDEX

Return to Cover Page