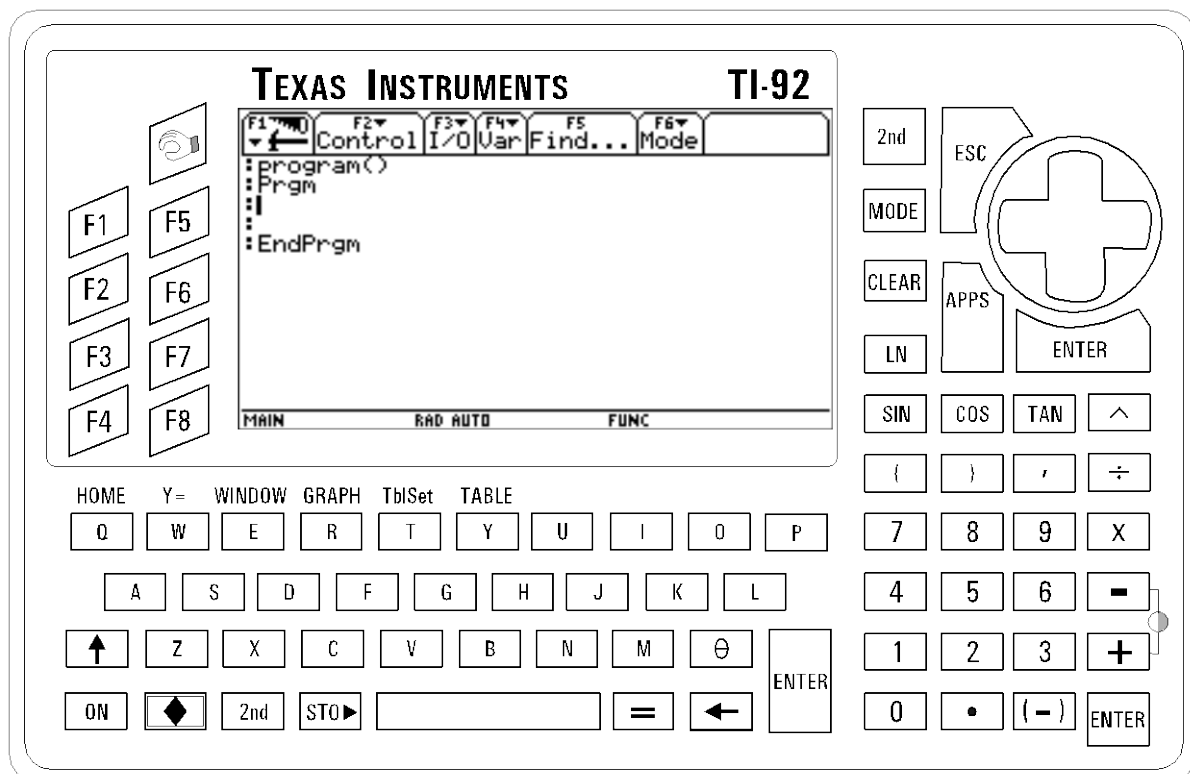


# Problem solving and Program design using the TI-92



**A.J. Marée**  
**M.H.A. van Dongen**

February 01, 2000



**Copyright © 1999 by A.J. Marée and M.H.A. van Dongen.**

**All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the writer A.J. Marée. Printed in the Netherlands.**

**ISBN 90-.....**



# CONTENTS

<b>PREFACE.....</b>	<b>V</b>
<b>1. INTRODUCTION .....</b>	<b>1</b>
1.1 STARTING A SIMPLE PROGRAM.....	1
<b>2. PROGRAM STYLE .....</b>	<b>3</b>
2.1 CLEAR SCREENS AND RESET MEMORY.....	3
2.2 CREATING A NEW FOLDER .....	4
2.3 COMMENTS .....	5
2.4 INDENTATION .....	6
2.5 CREATING A NEW PROGRAM .....	6
2.6 DEBUGGING A PROGRAM.....	8
2.7 ENTERING A GRAPHING PROGRAM .....	10
2.8 LISTING THE PROGRAMS YOU HAVE CREATED .....	12
2.9 SUMMARY OF COMMANDS .....	13
2.10 PRACTICAL PROBLEMS.....	14
<b>3. PROGRAM STRUCTURE.....</b>	<b>15</b>
3.1 DRAWING A SPECIFIC OBJECT .....	15
3.2 MAKING A PROGRAM MORE FLEXIBLE .....	16
3.3 COURTEOUS PROGRAMMING: GETMODE & SETMODE .....	20
3.4 SUMMARY OF COMMANDS .....	21
3.5 PRACTICAL PROBLEMS.....	23
<b>4. TOP-DOWN DESIGN &amp; PROGRAM DESIGN .....</b>	<b>25</b>
4.1 SOFTWARE DEVELOPMENT METHOD.....	25
4.2 TOP-DOWN DESIGN.....	28
4.2.1 <i>Drawing a House: An example</i> .....	28
4.3 SUMMARY OF COMMANDS .....	34
4.4 PRACTICAL PROBLEMS.....	35
<b>5. SELECTION CONTROL STRUCTURES .....</b>	<b>37</b>
5.1 THE IF...THEN...ENDIF STATEMENT .....	37
5.1.1 <i>Order a pair of data values: An example</i> .....	37
5.1.2 <i>Is the number an integer? : An example</i> .....	38
5.2 THE IF...THEN...ELSE...ENDIF STATEMENT .....	39
5.2.1 <i>The absolute value of a number: An example</i> .....	39
5.2.2 <i>Rental Car Pricing: An example</i> .....	40
5.3 THE IF...THEN...ELSEIF...ENDIF STATEMENT .....	40
5.4 TOP-DOWN PROGRAM DESIGN .....	41
5.4.1 <i>The ABC Formula: An example</i> .....	43
5.4.2 <i>Assigning Exam Scores to Letter Grades: An example</i> .....	48
5.5 SUMMARY OF COMMANDS .....	50
5.6 PRACTICAL PROBLEMS.....	52
<b>6. REPETITION CONTROL STRUCTURES .....</b>	<b>55</b>
6.1 THE FOR...ENDFOR STRUCTURE .....	55
6.2 THE LOOP...ENDLOOP STRUCTURE.....	56
6.3 THE WHILE...ENDWHILE STRUCTURE .....	57
6.4 USING LOOPS TO ACCUMULATE A SUM AND AVERAGE.....	58
6.4.1 <i>Using a For...EndFor Loop</i> .....	58
6.4.2 <i>Using a Loop ... EndLoop Loop</i> .....	59
6.4.3 <i>Using a While ... EndWhile Loop</i> .....	59
6.5 NESTED LOOPS .....	60
6.5.1 <i>Create and Fill a Matrix: An example</i> .....	61
6.5.2 <i>Print Out Zeros of an Expression: An example</i> .....	64
6.6 SUMMARY OF COMMANDS .....	65

6.7	PRACTICAL PROBLEMS.....	67
<b>7.</b>	<b>FUNCTIONS, SUBROUTINES, PROGRAMS AND PARAMETERS.....</b>	<b>71</b>
7.1	FUNCTIONS .....	71
7.1.1	<i>Create an Absolute Value Function: An Example .....</i>	<i>73</i>
7.1.2	<i>Create an Factorial Function: An Example.....</i>	<i>74</i>
7.2	INTRODUCTION TO PARAMETER LISTS.....	75
7.3	SUMMARY OF COMMANDS .....	78
7.4	PRACTICAL PROBLEMS.....	79
<b>8.</b>	<b>DATA TYPES.....</b>	<b>81</b>
8.1	REAL NUMBERS.....	81
8.2	EXPRESSIONS.....	81
8.3	STRING VARIABLES .....	82
8.3.1	<i>Create a Count Letters Program: An Example.....</i>	<i>83</i>
8.4	LIST VARIABLES.....	84
8.4.1	<i>Findkey, a List Variables Program: An Example .....</i>	<i>85</i>
8.5	DATA VARIABLES.....	87
8.6	MATRIX VARIABLES.....	88
8.6.1	<i>Matrix multiplication: An Example.....</i>	<i>89</i>
8.7	DATA TYPES AND DATABASES .....	89
8.7.1	<i>Access information in a NAC database: An Example .....</i>	<i>91</i>
8.8	SUMMARY OF COMMANDS .....	97
8.9	PRACTICAL PROBLEMS.....	100
<b>9.</b>	<b>MENUS AND DIALOG BOXES.....</b>	<b>103</b>
9.1	DESIGNING MULTIPLE MENUS.....	103
9.2	CREATING DIALOG BOXES .....	105
9.3	CREATING CUSTOM AND POP-UP MENUS.....	108
9.4	SUMMARY OF COMMANDS .....	112
9.5	PRACTICAL PROBLEMS.....	115
<b>APPENDIX</b>	<b>.....</b>	<b>117</b>
<b>A.</b>	<b>TI-92 FUNCTIONS, INSTRUCTIONS AND COMMANDS .....</b>	<b>119</b>
<b>B.</b>	<b>TI-92 CHARACTER CODES .....</b>	<b>139</b>

## Preface

This textbook is intended for a basic course in problem solving and program design needed by scientists and engineers using the TI-92. The TI-92 is an extremely powerful problem solving tool that can help you manage complicated problems quickly. We assume no prior knowledge of computers or programming, and for most of its material, high school algebra is sufficient mathematica background. It is advised that you have basic skills in using the TI-92. After the course you will become familiar with many of the programming commands and functions of the TI-92.

The connection between good problem solving skills and an effective program design method, is used and applied consistently to most examples and problems in the text. We also introduce many of the programming commands and functions of the TI-92 needed to solve these problems. Each chapter ends with a number of practica problems that require analysis of programs as well as short programming exercises. All programs listed in this book are available for downloading from [http://www.fontys.nl/procesregeling/probleemanalyse/index\\_proban.htm](http://www.fontys.nl/procesregeling/probleemanalyse/index_proban.htm)

We would welcome comments for improving this text, because this is one of the first textbooks to problem solving and program design fully integrate in the TI-92. If you have constructive criticisms to help writing a second edition please write to the authors in Eindhoven or send an E-mail to [T.Maree@fontys.nl](mailto:T.Maree@fontys.nl) or [MHA.vanDongen@fontys.nl](mailto:MHA.vanDongen@fontys.nl).

We hope that you will enjoy your course in problem solving and program design using the TI-92.

Finally, we thank our colleagues Jan Jelle Claus and Lilian van Erk-Frijters who provided us with significant contributions and many helpful comments and suggestions which improved the presentation.

**Ir.ing. A.J. (Ton) Marée**

**Ir. M.H.A. (Martijn) van Dongen**





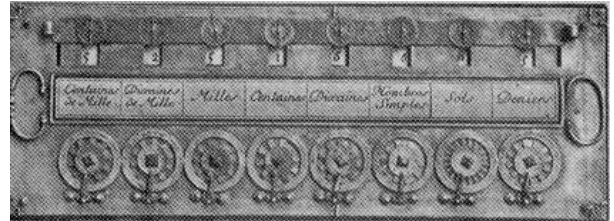
## 1. Introduction

### History

Blaise Pascal devised the first adding machine, a precursor of the digital computer, in 1642. He developed a mechanical calculator to speed arithmetic calculations for his father, a tax official. This device employed a series of ten-toothed wheels each tooth representing a digit from 0 to 9. The wheels were connected in such a way that numbers could be added to each other by advancing the wheels by a correct number of teeth. Numbers are dialed in on the metal wheels on the front of the calculator. The solutions appear in the little windows along the top.



**Pascal, Blaise** (1623-1662)  
French philosopher,  
mathematician, and physicist.



### Program

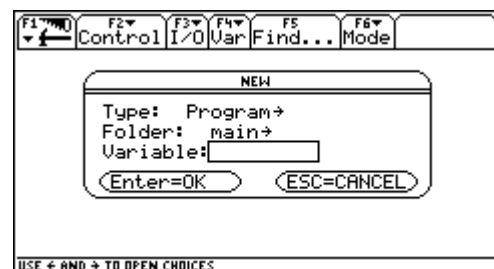
Since a computer is not able to think, it requires a program to do a task. A *program* is a sequence of instructions, which combined can perform any number of tasks.

Program design involves much more than simply writing a list of instructions. Problem solving is a crucial component of program design and requires a good deal of preplanning. Before writing a program to solve a particular problem, you must consider carefully all aspects of the problem breaking it down into small parts, and then writing commands that solve each of these smaller problems. You write programs in small chunks, following predefined steps. Certain parts must be placed in specific locations within a program and must follow certain rules.

### 1.1 Starting a Simple Program

#### The Program Editor

The mechanics of entering a program as a source file, translating it to machine language, and executing the machine language program differ on each computer system. In this text, we use the TI-92 pocket computer. The TI-92 provides an integrated programming environment, which means that you will be able to create, edit, compile and execute programs from within the TI-92.



You enter and edit programs in the Program Editor.

1. Press **[APPS]** to start a new program in the Program Editor, select **7:Program Editor**, and then select **3:New**.
2. Choose **Program** as the **Type**.

### Naming a Program

1. Press **[↓]** **[↓]** to move past the **Folder** item to the **Variable** item.
2. Type **hello** as the name of the new program variable.
3. Press **[ENTER]** to highlight the name.
4. Press **[ENTER]** again to accept all the choices in the dialog box.



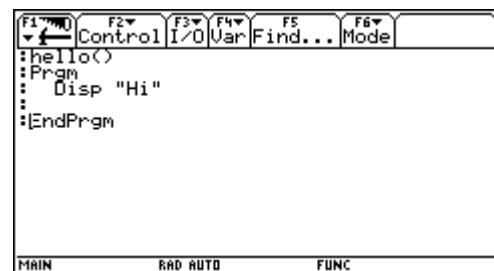
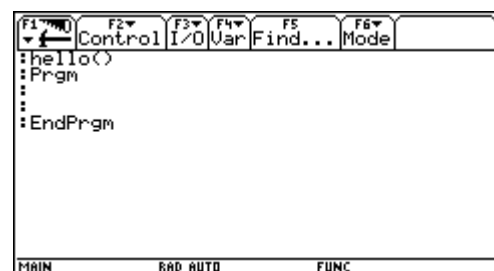
### Entering the Program Code

You are now in the Program Editor. The name of the program, followed by parentheses (), is displayed at the top of the program template, just below the menu items. The template also provides the colons (:) that begin every program line, the first statement, **Prgm**, and last statement, **EndPrgm**, every program requires these two statements.

Always start entering your program at the cursor, which is placed, between the **Prgm** and **EndPrgm** statements.

Type in the program line, the - Disp "Hi" - statement as shown to the right.

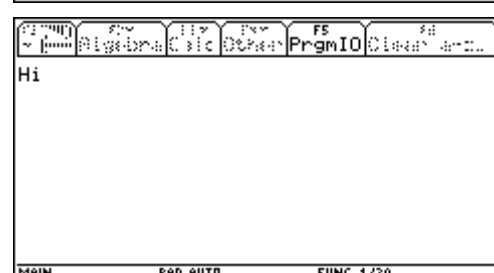
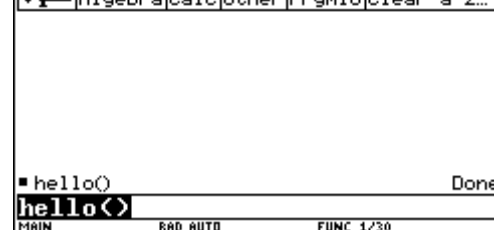
To type the quote marks, press **[2nd]** **[L]**.



### Executing the Program

1. Press **[♦]** **[HOME]** or **[2nd]** **[QUIT]** to return to the Home screen.
2. Press **[CLEAR]** to clear the entry line, if necessary
3. Type **hello()** and press **[ENTER]**.

The PrgmIO (Program Input/Output) screen is displayed with the word Hi on the last line of information. This line should be at the top of the PrgmIO screen (if you did not use the TI-programming facilities before).



## 2. Program Style

Program style refers to many aspects of programming-the formatting of various statements, comments, indentation of blocks, and so on. And just as there is no universal style for writing English, there is no universal style for writing programs. However, some general rules apply.

Good program style promotes the writing of programs that work and are easy to maintain and modify.

Don't fall into the trap of quickly typing in your program, with good intentions to edit it later on. Instead, type your programs using good program style from the very beginning.

### 2.1 Clear Screens and Reset Memory

Before beginning each TI-92 activity you are recommended to clear the HOME-screen the Y=Editor, any user-defined variables and partially reset the memory.

#### Clear the HOME-screen

Press: **[F1]** select 8:Clear Home **[ENTER]** **[CLEAR]**

#### Clear the Y=screen

Press: **[◀][Y=][F1]** select 8:Clear Functions **[ENTER]**

#### Clear (user defined) 1-character variables

Press: **[◀][HOME][F6][ENTER]**

#### The MEMORY screen

Press: **[2nd][MEM]** to show the memory screen. This screen shows how your TI-92's memory is being used. The screen shows the amount of memory (in bytes) used by each variable type and the amount of free memory. You can also use this screen to reset the memory.



## Resetting the Memory

1. Press: **[F1]** (from the MEMORY-screen)
2. Select the applicable item

### 1:All

Deletes all user-defined variables, functions, and folders; resets all system variables and modes to their original factory settings.

### 2:Memory

Deletes all user-defined variables, functions, and folders. This does not affect system variables (xmin, ymin, etc.) or mode settings.

### 3:Default (advised)

Resets all system variables and modes to their original factory settings. This does not affect any user-defined variables, functions, or folders.



## 2.2 Creating a new folder

### Using Folders

Folders give you a convenient way to manage programs and variables by organizing them into related groups. For example, you can create separate folders for different TI-92 applications. The TI-92 has one built-in folder named MAIN, and by default all variables are stored in that folder. By creating additional folders, you can store programs, independent sets of user-defined variables and functions apart from each other.

System variable or a variable with a reserved name are independent of the folder and are not shown in the [VAR-LINK]-screen (Y1(x), xmin, xmax, Y= Editor functions etc.).

User-defined variables however are dependent of the folder and can therefore have different values in different folders.

The user-defined variables in one folder are independent of the variables in any other folder. Therefore, folders can store separate sets of variables with the same names but different values.

**Note:** User-defined variables are stored in the "current folder" unless you specify otherwise. You can access a user-defined variable that is not in the current folder. Specify the complete *pathname* instead of only the variable name.

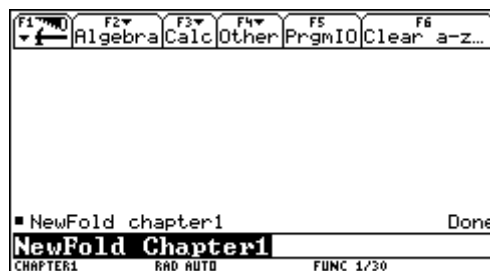
A pathname has the form:

*foldername\variableName.*



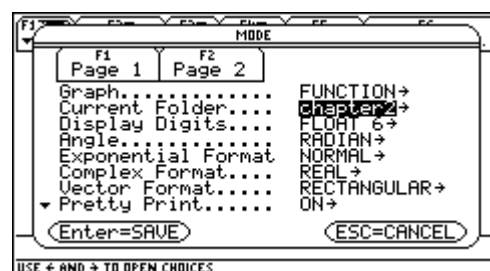
You can **create a new folder** from:

1. the Home screen  
Enter the NewFold command:  
**NewFold** *foldername*  
this new folder "*foldername*" is set automatically as the current folder.
2. the [VAR-LINK]-screen (Refer to your TI-92 Guidebook).



You can set the current folder from:

1. the Home screen  
Enter the setFold function: **setFold** (*foldername*)  
setFold is a function, which requires you to enclose the folder name in parentheses ().
2. the MODE Dialog Box; Press: [MODE] select Current Folder etc.. (Refer to your TI-92 Guidebook).



## 2.3 Comments

Programmers make a program more readable by using comment to describe the purpose of the program, the use of the identifiers, and the purpose of each program step.

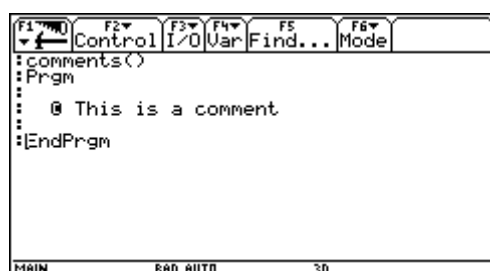
Before implementing each step in the initial algorithm, you should write a comment that summarizes the purpose of the algorithm step. This comment should describe what the step does. Comments help the reader of the program understand how it works.

Commenting takes a little effort, but it is an invaluable aid in debugging, maintaining, and enhancing your programs. Someone other than yourself may have to modify your program someday: If your program didn't have comments, it would be like cooking a gourmet meal from a recipe that lists all the ingredients but gives no instructions on how to combine them.

Comments begin with a **⓪**. Any information to the right of the **⓪** is not executed when you run the program.

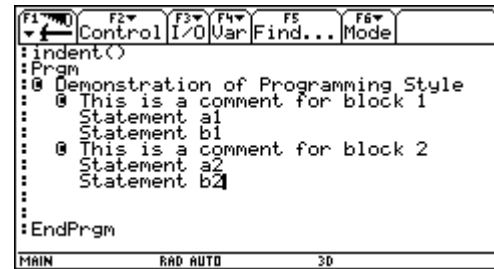
**Hint:** to type **⓪**, press [2nd] X.

Begin every program block of associated statements with a comment line.



## 2.4 Indentation

Indent one or two spaces, after the comment-line of every block of statements, which belong together. Make it a habit to indent and align associated statements. This save you hours of thumbing through printouts, trying to found out which statements belong together. It is easy to read and understand, (*especially for someone other than yourself*).



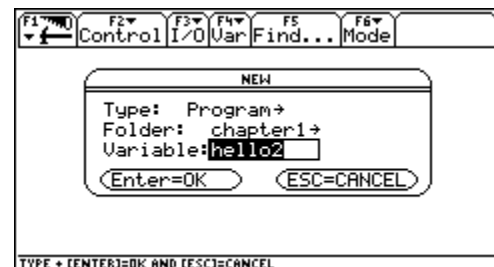
## 2.5 Creating a new program

To create a new program, you begin:

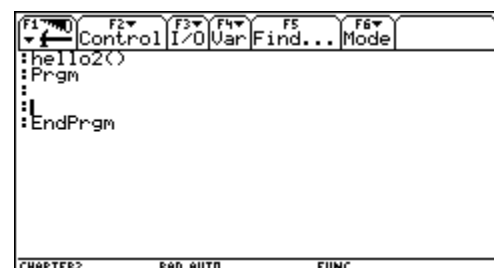
1. **Clear** the HOME-screen the Y=Editor, any user-defined variables and reset the memory.
2. **Make a new folder** (from the Home screen)
3. **Start a new Program in the Program Editor**  
You enter and edit programs in the Program Editor. Press [APPS] to start a new program in the Program Editor, select **7:Program Editor**, and then select **3:New**.  
Choose **Program** as the **Type**.



4. **Naming the Program**  
Press [DOWN] [DOWN] to move past the **Folder** item to the **Variable** item.  
Type **hello2** as the name of the new program variable.  
Press [ENTER] to highlight the name.  
Press [ENTER] again to accept all the choices in the dialog box.



5. **Entering the Program Code**  
You are now in the Program Editor. The name of the program, followed by parentheses (), is displayed at the top of the program template, just below the menu items. The template also provides the colons (:) that begin every program line, the first statement, Prgm, and last statement, EndPrgm, every program requires both of which.  
Always start entering your program at the cursor, which is placed, between the Prgm and EndPrgm statements.



Type in the program line, the  
Disp "Hi again"  
statement as shown to the right.  
To type the quote marks, press **[2nd]** **L**

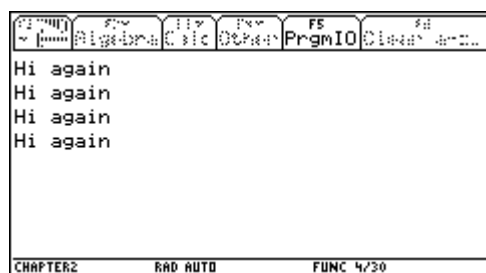


## 6. Running the Program

Press **[♦]** **[HOME]** or **[2nd]** **[QUIT]** to return to the Home screen.

Press **[CLEAR]** to clear the entry line, if necessary.

Type **hello2()** and press **[ENTER]**.



The PrgmIO (program input/output) screen is displayed with the word Hi on the last line of information. This line should be at the top of the PrgmIO screen. Run the program again and again.

Return to the Home screen by pressing

**[♦]** **[HOME]**.

Notice that **hello2()** is still in the entry line.

Once again, "Hi again" is displayed on the PrgmIO screen. This screen looks a lot like the Home screen; however, you cannot access any of the menu items at the top of the screen (except PrgmIO), and there is no entry line. You may sometimes mistake this screen for the Home screen. The Home screen is the screen where you ran the hello2() program from.

## 7. Changing the program

### Clear the PrgmIO Screen.

Press **[APPS]**, select **7:Program Editor**, and then select **1:Current**.

Move the cursor to the end of **Prgm** and press **[ENTER]** to insert a new blank line.

With the cursor at the beginning of this new blank line, press the space bar twice for the proper indentation and type **ClrIO**. This statement Clears the PrgmIO Screen.

Run the program again and again.



### Clear the Home Screen

Press **[APPS]**, select **7:Program Editor**, and then select **1:Current**. Move the cursor to the

end of Prgm and press **[ENTER]** to insert a new blank line.

With the cursor at the beginning of this new blank line, press the space bar twice for the proper indentation and type **ClrHome**. This statement Clears the Home Screen.

Run the program and go back to the Home Screen.

## 2.6 Debugging a Program

The TI-92 is designed to help you find and correct errors in your programs. This process is called "debugging." Debugging is the process of finding and correcting errors ("bugs").

When you try to run a program that contains a syntax error, such as a misspelled command name or a missing quotation mark, the TI-92 displays an error message like the one shown to the right. To return to the program and correct the error, press **[ENTER]**.

The program is displayed with the cursor on the line where the error occurred or where the program has gone wrong.

You can trace backward to find the syntax error. Correct the error, return to the Home screen, and run the program again.

### Example 1: A missing quotation mark"

Remove the last quotation mark in the statement:

Disp "Hi again

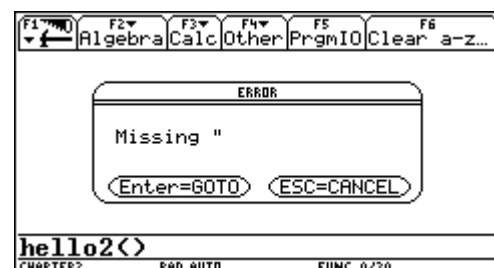
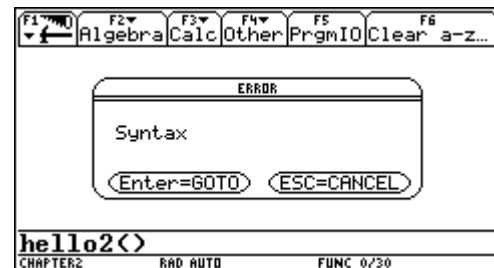
Run the program

The TI-92 displays an error message like the one shown to the right.

To return to the program and correct the error, press **[ENTER]**.

The program is displayed with the cursor on the line where the error occurred or where the program has gone wrong.

Unfortunately, error messages are often difficult to interpret and are sometimes misleading. Even the cursor position should be regarded only as indicating the approximate position of the error. For example:





"Correct the error" by typing the quotation mark behind EndPrgm where the cursor is displayed .

Run the program again.

"Correct the error" by typing **EndPrgm** to the end of the program.

Run the program again.

The program contains no syntax errors!

The PrgmIO screen is displayed with the line ("string"): Hi again.↵↵EndPrgm

Note: A string is a sequence of characters enclosed in quotes that allows the program to display information or prompts the user to perform an action.

↵: [ENTER]-symbol.

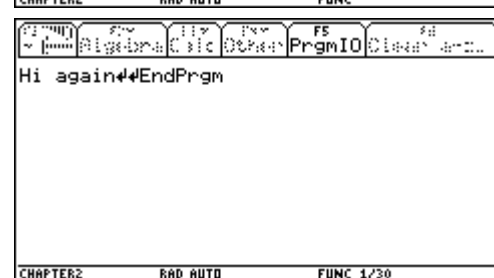
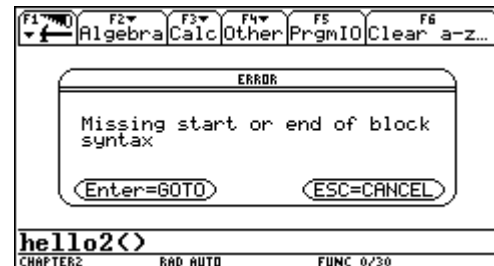
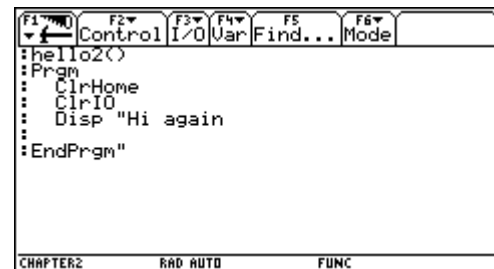
### Example 2: The first statement "Prg" instead of "Prgm"

Change the first statement "Prgm" to "Prg" and run the program. The TI-92 displays an error message. Return to the program.

The program is displayed with the cursor on the line to the end of the **ClrHome** statement "where the error occurred".

### Conclusion:

When you try to find the location where the error occurred. It is mostly not where the cursor stands. **Therefore trace backwards to find the (syntax) error!!**



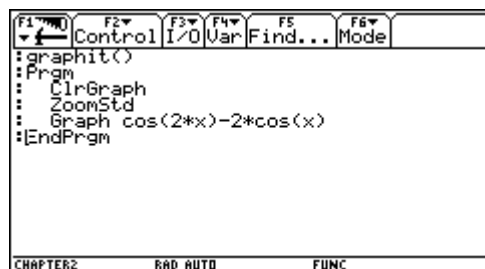
## 2.7 Entering a Graphing Program

You can write programs that use the graphing capabilities of the TI-92.

To start the new program in the Program Editor, press  $\blacktriangleright$ , select 7:Program Editor, and then select 3:New. Press  $\blacktriangleright$ , and type **graphit** as the name of the new program variable.

Press  $\text{ENTER}$  to store the program name and display the new program template.

Type the program lines as shown to the right, making sure to indent for readability. Press  $\text{ENTER}$  at the end of  $\text{ENTER}$  each line.



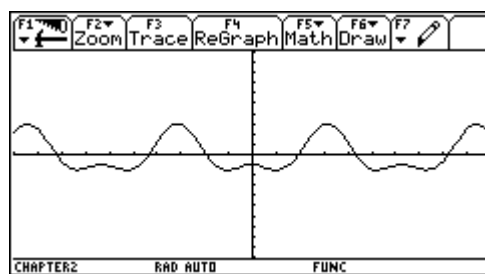
This program graphs the function given by the expression  $\cos(2x) - 2\cos(x)$ . The **ClrGraph** statement clears any functions that were graphed with the **Graph** command. The **ZoomStd** statement ensures that the function will be graphed in the standard viewing window. You must, however, set the graph mode to **FUNCTION** for the program to work correctly. To do this, press the  $\blacklozenge$  [HOME] to return to the Home screen, press  $\blacktriangleright$  to check the graph mode, and set the mode to **FUNCTION**, if necessary. You exit the Mode dialog box by pressing  $\text{ESC}$  (or press  $\text{ENTER}$  twice if you make a change).

Run the program. Type **graphit()** on the entry line of the Home screen and press  $\text{ENTER}$ .

The graph of the function whose expression is  $\cos(2x) - 2\cos(x)$  is displayed in the standard viewing window.

**Hint:** If you have previously defined a function or statistics plot, press  $\blacklozenge$  [Y=] to display the Y= Editor. Then move to each selected function or plot and press  $\text{F4}$  to deselect it. This ensures that only the graph for the function in your program is displayed.

To turn off All functions, press  $\text{F5}$ :All and then select 1:All Off or deselect all functions from the Home screen; press  $\text{F4}$ :Other and then select 8:FnOff command. You can also use the **FnOff** command in your (graphit) program.



### The Mode

Press **[MODE]**, and change the Graph mode to **3D**

Run the program.

The program now returns the graph shown to the right.

Another example.

Press **[MODE]**, and change the Graph mode to **POLAR**

**POLAR**

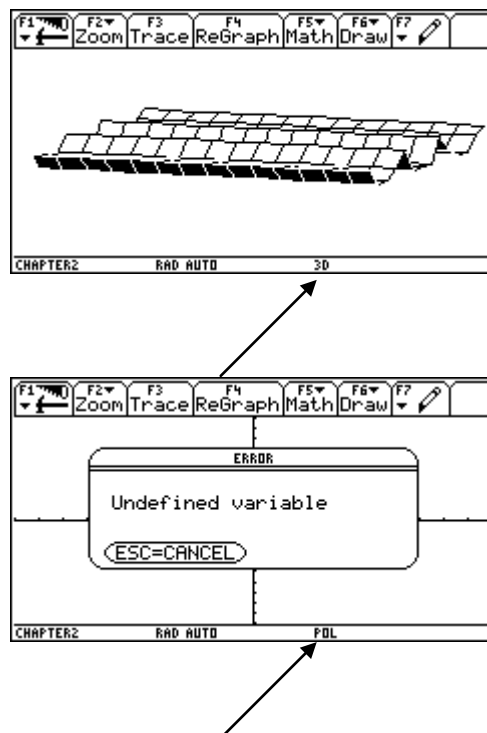
Run the program.

The program now returns an error due to the incorrect graph mode.

The program expects the Graph mode

"FUNCTION" (or 3D) but uses POLAR mode and therefore returns an error.

You can improve your program by including a line that sets the graph mode to FUNCTION.



### Using the `setmode(" Graph", " FUNCTION")` Command

Edit the current program to do this.

Press **[APPS]**, select 7:Program Editor, and then select 1:Current.

Move the cursor to the end of `ZoomStd` and press **[ENTER]** to insert a new blank line.

With the cursor at the beginning of this new blank line, press the space bar twice for the proper indentation.

Now press **[F6]**. The **Mode** menu is displayed.

Press **[F1]** to display the Graph submenu.

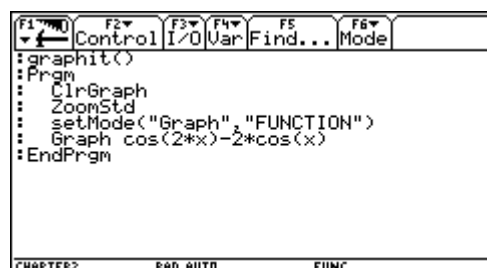
Select 1:FUNCTION. The command:



`setMode(" Graph", " FUNCTION")`

is inserted at the cursor into the new line.

Now you are assured that the graph mode is correct without checking it or setting it yourself outside the program.



## 2.8 Listing the programs you have created

You can display a list of the programs you have created in two ways.

1. Press **[APPS]**, select 7:Program Editor, and select 2:Open. A dialog box is displayed. The first program name appears next to the Variable item followed by an indicating that there are more programs.



Press **[◀]** and then **[▶]**. The names of your programs are listed in alphabetical order.

To select a program, move the cursor to highlight its name and press **[ENTER]**. The program is displayed in the Program Editor.



2. Displaying the VAR-LINK screen by pressing **[2nd][VAR-LINK]**. By default, the VAR-LINK screen lists all user-defined variables in all folders and with all data types.

To select a program, type a letter repeatedly to cycle through the names that start with that letter, or move the cursor to highlight its name.

Press **[F6]** to show the contents of the program, but you cannot edit the contents from the VAR-LINK-screen.

Press **[ENTER]** to return to the HOME-screen, ready to run the program.

## 2.9 Summary of commands

<b>ClrGraph</b>	Clears any functions or expressions that were graphed with the Graph command (See Graph on page 127) Any previously selected Y= functions will be graphed the next time that the graph is displayed.
<b>ClrHome</b>	Clears the Home Screen; all items stored in the <b>entry()</b> and <b>ans()</b> Home screen history area. Does not clear the current entry line. While viewing the Home screen, you can clear the history area by pressing <b>[F1]</b> and selecting 8:Clear Home.
<b>ClrIO</b>	Clears the Program I/O Screen.
<b>Disp</b>	Displays the current contents of the Program I/O screen.
<b>Disp</b> [ <i>exprOrString1</i> ][, <i>exprOrString2</i> ]	Displays each expression or character string on a separate line of the Program I/O screen. <u>For example:</u> Disp "Hi again" <b>[ENTER]</b> returns: Hi again Disp sin( $\sqrt{6}$ ) <b>[ENTER]</b> returns: 1/2 If Pretty Print = ON , expressions are displayed in pretty print.
<b>FnOff</b>	Deselects all Y= functions for the current graphing mode.
<b>FnOff</b> [ <i>1</i> ] [, <i>2</i> ] ... [,99]	Deselects the specified Y= functions for the current graphing mode. <u>For Example:</u> <b>FnOff 1,3</b> deselects Y1 (x) and Y3(x).
<b>FnOn</b>	Selects all Y= functions that are defined for the current graphing mode.
<b>FnOn</b> [ <i>1</i> ] [, <i>2</i> ] ... [,99]	Selects the specified Y= functions for the current graphing mode.
<b>Graph</b> <i>expression</i> [, <i>var</i> ]	Graphs the requested <i>expression</i> / <i>function</i> using the current graphing mode. If you omit an optional <i>var</i> argument, <b>Graph</b> uses the independent variable of the current graphing mode. <u>For example:</u> Graph sin(t),t <b>[ENTER]</b>
<b>NewFold</b> <i>foldername</i>	Creates a user-defined folder with the name <i>foldername</i> , and then sets automatically the current folder to that folder. After you execute this instruction, you are in the new folder.
<b>setFold</b> ( <i>newfolderName</i> )	$\Rightarrow$ <i>oldfolderString</i> Returns the name of the current folder as a ( <i>oldfolder</i> ) <i>string</i> and sets <i>newfolderName</i> as the current folder. SetFold is a function, which requires you to enclose the newfoldername in parentheses (). <b>Note:</b> The folder newfolderName must exist.

**setMode()**

Sets mode to the new settings. In the Program Editor **[F6]**: Mode menu (see also setMode settings on page 135).

**ZoomStd**

Sets the window variables to the standard values, and then updates the ZoomStd viewing window.

Standard values for function graphing:

x:[-10,10,1], y:[-10,10,1] and xres=2

**2.10 Practical problems*****Problem 1***

Create a program, which will display a random number between 0 and 10 on the screen.

Note: use rand().

***Problem 2***

Modify the program made in problem 1 to change the look of the number to seven decimals (for example 20 → 20.0000000)

Hint: Display Digits in FIX Format.

### 3. Program structure

In the previous chapter, we looked at some statements and put them to work in a simple program. This and the next chapters cover the rules governing the structure of programs in more detail.

Let's quickly review what we've learned about program structure so far. A program consists of two distinct parts: the program heading and the statement part. The program heading's primary function is to name the program.

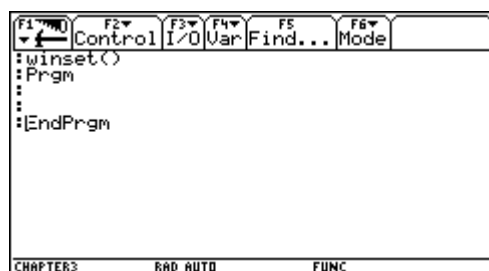
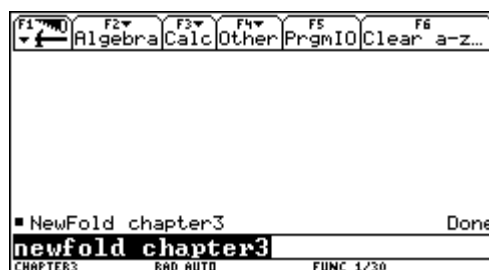
The statement part is one or more program statements that describe the actual work to be performed—such as adding numbers, making assignments, and printing information to the program I/O screen. The statements appear between the reserved words Prgm and EndPrgm. This part of the program is also called the main program.

#### 3.1 Drawing a specific object

The following instructions tell you how to draw a box. You begin with a program that sets the values of the Window variables. Finally you enter a program that draws a box using the specified Window variable values.

#### Setting the Values of the Window Variables

1. First, clear the HOME-screen the Y=Editor, any user-defined variables and reset the memory to the default settings.
2. Second create a new folder chapter3.
3. To start a new program in the Program Editor, press **[APPS]**, select **7:Program Editor**, and then select **3:New**. Press **⓪** and type **winset** as the name of the new program variable.
4. Press **[ENTER][ENTER]** to store program name and display the new program template.
5. Type the program lines as shown to the right. (Notice the indentation pattern.) Press **[ENTER]** at the end of each line.  
To type **⇒**, press **[STO▶]**.
6. The **⓪** symbol indicates a program comment, which you can use to explain or describe



```
winset()
Prgm
" Set the Window Variables
0⇒xmin " Stores 0 in xmin
15⇒xmax
0⇒ymin
10⇒ymax
EndPrgm
```

various aspects of the program. Any information to the right of the ● is not executed when you run the program.

**Hint:** To type ●, press [2nd] X.

7. After you have entered all the program lines, your screen should look like the one to the right. Now press [◆][HOME].
8. To run the program, type **winset()** and press [ENTER].

Although **Done** is displayed on the Home screen, nothing seems to have happened. This program changes the Window variable values for function graphing. To see the results, press [◆][WINDOW]. The values for **xmin**, **xmax**, **ymin**, and **ymax** should match the values in the program.



### Drawing the Box

1. Start a new program in the Program Editor, and name it **drawbox**.
2. Enter the program lines as shown to the right, making sure to indent for readability.
3. Press [◆][HOME] Type **drawbox()** and press [ENTER].

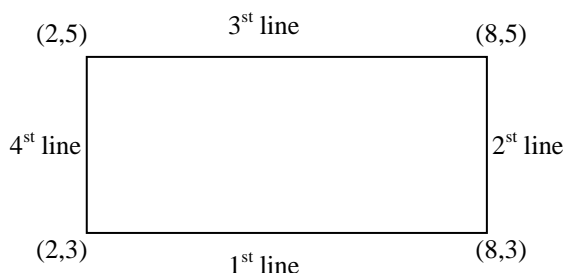
Where is the box located on the screen? How have the menu items at the top of the screen changed? How would you get back to the Home screen?

The box is located toward the lower left part of the screen. The menu items now relate to graphing since you are on the Graph screen. You can return to the Home screen by pressing [◆][HOME].

```

drawbox()
Prgrm
" Set the Window Variables
0→xmin " Stores 0 in xmin
15→xmax
0→ymin
10→ymax
ClrGraph " Clears the Graph Screen
ClrDraw " Clears the Drawings
setMode("Graph","FUNCTION")
" Draw the Box
Line 2,3,8,3 " Draws a Line Segment
Line 8,3,8,5
Line 8,5,2,5
Line 2,5,2,3
EndPrgrm

```



### 3.2 Making a program more flexible

When you make a program flexible, it can be used to solve a greater variety of problems. This next section shows you how to add program code that:

- lets the user choose the exact location of the box.



- lets the user choose the Window variable values for the screen and choose the exact location of the box.
- lets the user choose the size of the box and choose the exact location of the box.

Before you begin the new program, compare it to the drawbox program.

- You can copy lines from one program to another.  
The TI-92 allows you to cut, paste, and copy characters with  $\diamond X$ ,  $\diamond V$ , or  $\diamond C$ . You can also use the  $\square$  CLEAR or the  $\leftarrow$  keys to help you edit on the entry line.

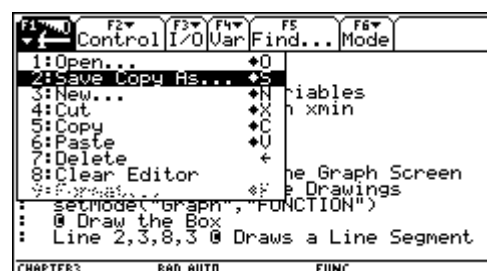
- You can save the current program, for example drawbox, under a different name then open the new program and make a few changes.  
Choose  $\square$  F1 select 2:Save Copy As... and type for example placebox, open the "new" program placebox and make the necessary changes.

### The Cut, Copy, and Paste Functions

For example,

- 1) Press  $x \square 2 \square 3x \square 4 \square$  ENTER
- 2) To cut  $-3x$  from this expression, move the cursor to the right of 2. Press and hold down the shift key  $\square$  and press three times to highlight  $-3x$ . Now press  $\diamond X \square$  ENTER
- 3) On a cleared entry line, press  $\diamond V$  to retrieve this expression from the buffer where it was stored.

Note: You saved the expression  $-3x$  when you pressed  $\diamond X$ . The delete ( $\diamond X$ ), paste ( $\diamond V$ ), and copy ( $\diamond C$ ) features of the TI-92 are similar to the cut and paste features on most computer word processors.



### Let the user choose the location of the box

1. Start a new program in the Program Editor and name it **placebox**.
2. Enter the program lines as shown to the right.  
*Note: In this book, when a line is too long to fit on the screen as in the eighth program line here, it is continued and indented on the next line. However, you should type the line without breaking it and let the calculator wrap the line.*
3. Run the **placebox** program.
4. Enter the x- and y-coordinates of the lower left corner when requested. Think about the currently defined Graph screen and where you might locate the corner in such a way that the entire box is displayed.  
Was the box displayed where you expected?

```
placebox()
Prgm
" Draw a Box at Location You Choose
" Set the Window Variables
0→xmin " Stores 0 in xmin
15→xmax
0→ymin
10→ymax
" Choose the Location
Input "Enter x-coordinate of lower
left corner",x
Input "Enter y-coordinate",y
ClrDraw " Clears the Drawings
setMode("Graph","FUNCTION")
" Draw the Box
Line x,y,x+2,y " Draws a Line
Line x+2,y,x+2,y+2
Line x+2,y+2,x,y+2
Line x,y+2,x,y
EndPrgm
```

### Letting the user choose the window variable values and the box location

Let us continue to extend this example. Suppose you would like to set up a specific Graph screen while the program is executing and then draw the box on that screen. Once again, think about how you can accomplish this before you enter the program lines.

You need a request to the user to enter each of the main Window variable values: **xmin**, **xmax**, **ymin**, and **ymax**. Then you must be able to enter the desired values.

The following program extends the **placebox** program. As you enter the program, you can either type the commands **Input**, **Line**, and **ClrDraw** from the keyboard or paste them in from the appropriate menu. For example, to paste the **Input** command in your program, place the cursor at the appropriate position, press [F3], and then select **3:Input**.

- 1) Start a new program in the Program Editor and name it **chuzwin**.
- 2) Enter the program lines as shown to the right. Notice that the four Input lines for the Window variable values are almost identical. Therefore, after typing the first one, you could follow these steps to copy and edit it instead of typing the other three lines.
  - a) Type the first Input line and then move the cursor to the beginning of the line.
  - b) Hold down [↑] and press [↻] once. This highlights the line.
  - c) Press [♦] C to copy the line.
  - d) Move the cursor to the beginning of the next line.
  - e) Press [♦] V to paste the copy of the line here.
  - f) Edit the line to match the program at the right.
  - g) Repeat these steps for the other two **Input** lines.

Enter the remaining program lines.
- 3) Return to the Home screen, and run the chuzwin program.

```
chuzwin()
Prgm
" A Box at a Location You Choose
" Choose your Window Values
Input "Enter xmin value",xmin
Input "Enter xmax value",xmax
Input "Enter ymin value",ymin
Input "Enter ymax value",ymax
" Choose the Location
Input "Enter x-coordinate of lower
left corner",x
Input "Enter y-coordinate",y
ClrDraw " Clears the Graph Screen
" Draw the Box
Line x,y,x+2,y " Draws a Line
Line x+2,y,x+2,y+2
Line x+2,y+2,x,y+2
Line x,y+2,x,y
EndPrgm
```

## Letting the User Choose Box Size and Location

Are there other ways to extend the program? Perhaps you would like to be able to choose the height and width of the box along with the placement of the bottom left corner of the box. In addition, suppose you would like to ensure that the entire box actually appears in the Graph screen.

Again, think about how you can accomplish these tasks before typing in the program.

1. Start a new program in the Program Editor and name it **chuzbox**.
2. Enter the program lines as shown to the right.
3. Return to the Home screen, and run the **chuzbox** program.

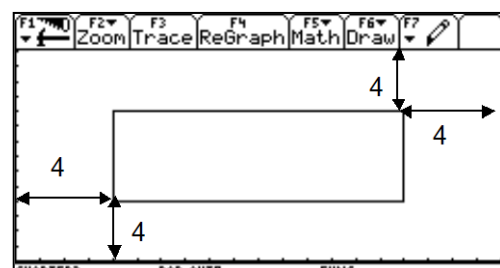
Experiment with any values you desire for the width and height of the box. Do the same for the location of the lower left corner.

Was the entire box displayed in the Graph screen? Can you choose values for the dimensions of the box and location of the corner whereby only a portion of the box is drawn in the Graph screen? Try several combinations.

By thinking about what the program does, you can be sure that the entire box is always displayed in the Graph screen since the Window variable values are determined by the numbers you put in enter.

```
chuzbox()
Prgm
" Choose a Box Size and its Location
" Program also Determines View Window
ClrIO " Clears the IO-Screen
" Choose your Box Size
Input "Enter width of box",x
Input "Enter height of box",y
" Choose the Location
Input "Enter x-val of low left
corner",xx
Input "Enter y-val of corner",yy
" Set the Window Values
xx-4→xmin
xx+x+4→xmax
yy-4→ymin
yy+y+4→ymax
ClrDraw " Clears the Graph Screen
" Draw the Box
Line xx,yy,xx+x,yy " Draws Line
Line xx+x,yy,xx+x,yy+y
Line xx+x,yy+y,xx,yy+y
Line xx,yy+y,xx,yy
EndPrgm
```

By setting the Window values as indicated, the Graph screen is always at a distances of 4 units from the box's edges



### 3.3 Courteous programming: **getMode** & **setMode**

If you need to change mode settings within your program, consider using the **getMode** and **setMode** functions to restore your TI-92's previous settings.

*(If you are using someone else's TI-92; you can leave the settings on the calculator as you found them.)* For an introduction to **getMode** and **setMode**, see page 126 and 135.

#### Preserving Mode Settings

The function **getMode** and **setMode** can be used together in a program to preserve the TI-92's mode settings. Use **getMode** to obtain the mode settings when the program begins and **setMode** to restore them before the program ends.

This program design as shown to the right illustrates the use of **getMode** and **setMode**: Call **getMode("ALL")** to obtain a list of the current mode settings as they exist before the program begins executing. In the example above, the list of mode settings is stored ( → ) in **prevmode**.

While the program is executing, it is free to change mode settings as needed for its proper operation.

Before the program terminates, restore the mode settings preserved in **prevmode** by calling **setMode(prevmode)**.

Because **prevmode** is a local variable, other programs cannot unintentionally change it. If other programs save and restore mode settings in this fashion, your program can call other programs as subroutines without concern that mode settings will not be restored correctly.

#### Temporary Mode Changes

The function **setMode** changes one or more mode settings. It returns a string (or list of mode/settings string pairs) indicating the previous mode state before the new mode settings are applied. This feature allows your program to temporarily change a mode setting, execute a few program statements, and then restore the previous mode setting.

```
ProgGet()
Prgm
  getMode("ALL")→prevmode
EndPrgm
```









```
ProgSet()
Prgm
  setMode(prevmode)
EndPrgm
```

```
Sample_1()
Prgm
  ClrIO
  Disp getMode("ANGLE")
  " Need degree mode temporarily
  setMode("ANGLE","DEGREE")»saveang

  Disp getMode("ANGLE")
  Disp saveang
  -----
  -----
  -----
  " restore previous angle mode
  setMode("ANGLE",saveang)

  Disp getMode("ANGLE")
EndPrgm
```

### 3.4 Summary of commands

	Deletes the character to the left of the cursor.
 C	Copies highlighted characters.
 V	Pastes highlighted characters.
 X	Cuts highlighted characters.
● [text]	press  X      A comment symbol ● lets you enter a remark in a program. When you run the program, all characters following ● are ignored; comments and are not executed.
<b>a</b> → <b>b</b>	<b>a</b>  <b>b</b> Stores the value on the left (a) in the variable on the right (b):
<b>Circle</b> x,y,r	Draws a circle with its center at (x,y) and radius of r.
<b>ClrDraw</b>	Clears the drawing from the screen.
<b>getMode</b> (modeNameString)	⇒ String If the argument is a specific mode name, returns a string containing the current setting for that mode. <u>For example:</u> <b>getMode</b> ("angle")  returns: "RADIAN"
<b>getMode</b> ("ALL")	⇒ String If the argument is "ALL", returns a list string pairs containing the settings of all the modes. <u>For example:</u> <b>getMode</b> ("ALL")  returns: { "Graph" "FUNCTION" "Display Digits" "FLOAT 6" "Angle" "RADIAN" "Exponential Format" "NORMAL" "Complex Format" "REAL" "Vector Format" "RECTANGULAR" "Pretty Print" "ON" "Split Screen" "FULL" "Split 1 App" "Home" "Split 2 App" "Graph" "Number of Graphs" "1" "Graph 2" "FUNCTION" "Split Screen Ratio" "1: 1" "Exact/ Approx" "AUTO" } <b>Note 1:</b> Your screen may display different mode settings. <b>Note 2:</b> If you want to restore the mode settings later, you must store the <b>getMode</b> ("ALL") result in a variable, and then use <b>setMode</b> (see page 135) to restore the modes.
<b>Input</b> [promptString,] var	pauses the program, displays <i>promptString</i> on the Program I/O screen, waits for you to enter an expression, and stores the expression in variable <i>var</i> . If you omit <i>promptString</i> , "?" is displayed as a prompt.
<b>Line</b> x1,y1,x2,y2	Draws a line between the points (x1,y1) and (x2,y2).
<b>Prgm</b> <b>EndPrgm</b>	Designates the beginning and end of a program.

**setMode**(*list*)

*list* contains pairs of keyword strings.

Sets mode all at once to the new setting. This is recommended for multiple-mode changes.

**setMode**(*var*)

Use **setMode**(*var*) to restore settings saved with **getMode** ("ALL ")→*var*.

For a listing of mode names and possible settings, see setMode on page 135.

### 3.5 Practical problems

Think about how you might solve the following exercises before you write the program lines. In addition, run each program to be sure it gives the desired results.

#### **Problem 1**

Write a program that calculates and displays the value of the combined resistance  $R_v$  of three parallel resistance's  $R_1$ ,  $R_2$  and  $R_3$ .

Request the user to enter each value of the resistances.

**Hint:** the formula for three parallel resistance is:  $\frac{1}{R_v} = \frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3} + \dots$

**Note:**  $R_1$  is a system variable.

#### **Problem 2**

Modify the **winset** program by setting the minimum values to the negatives of the maximum values. Run the program after you modify, it, and check to see that the Window variable values have changed to the values in your modified program.

To be checked from the Graph screen where the origin is in the center.

#### **Problem 3**

Modify the **drawbox** program in such a way that the width of the box is four times the height. The values you use should ensure that the entire box is displayed in the Graph screen.

#### **Problem 4**

Modify the **drawbox** program in such a way that the box is a square and within the box is drawn a circle attached to each side of the box.

#### **Problem 5**

Modify the **placebox** program in such a way that each side of the box is six units in length.

#### **Problem 6**

Modify the **chuzbox** program in such a way that the rectangle as well the diagonals of the rectangle are drawn in the Graph-screen.

#### **Problem 7**

Modify the **chuzbox** program in such a way that a circle with center at (a,b) and radius r is drawn and within the circle the axes attached to the circle.





## 4. Top-down Design & Program Design

### 4.1 Software Development Method

Program design is a problem solving activity. If you are a good problem solver, you have the potential to become a good program designer. One goal of this book is to help you improve your ability to solve problems.

We introduce a systematic approach to solving program design problems called the software development method and show you how to apply it.

#### **The Software Development Method:**

1. Specify the problem requirements
2. Analyze the problem
3. Design the algorithm to solve the problem
4. Implement the algorithm
5. Test and verify the completed program
6. Maintain and update the program.

The steps in the software development method are presented below in more detail.

#### **PROBLEM**

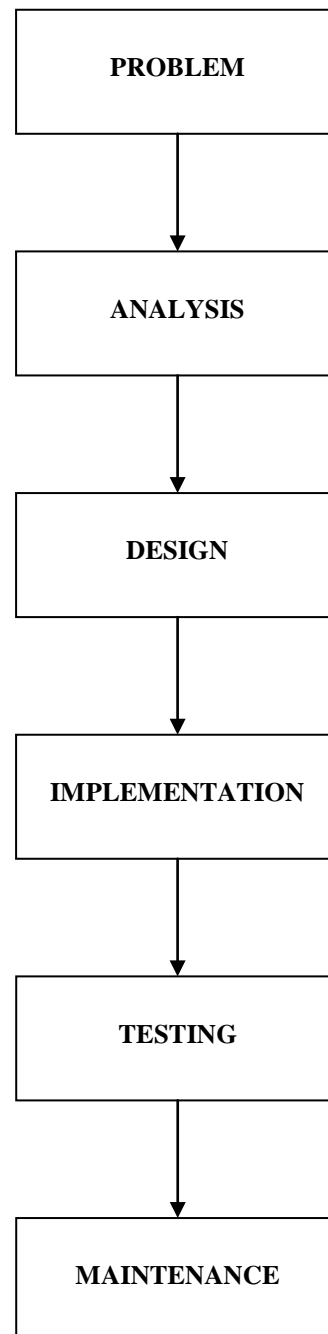
*Specifying the problem requirement* forces you to state the problem clearly and unambiguously and to gain a clear understanding of what is required for its solution. Your objective is to eliminate unimportant aspects and focus on the main and problem. This may not be as easy as it sounds. You may find you need more information from the person who posed the problem.

#### **ANALYSIS**

*Analyzing the problem* involves describing the problem in basic ingredients:

- (a) inputs, that is, the data you have to work with;
- (b) outputs, the desired results; and
- (c) any additional requirements or constraints the problem presents you with.

At this stage, you should also determine the required format in which the results should be displayed and develop a list of problem variables and their relationships. These relationships may be expressed as formulas.



If the steps 1 and 2 are not done properly, you will solve the wrong problem. Read the problem statement carefully, first to obtain a clear idea of the problem and second determine the input and outputs. You may find it helpful to underline phrases in the problem statement that identify the inputs and outputs, as in the following problem statement:

Determine the total cost of apples given the number of pounds of apples purchased and the cost per pound of apples.

Problem Inputs:

- quantity of apples purchased (in pounds)
- cost per pound of apples (in dollars per pound)

Problem Output

- total cost of apples (in dollars)

Once you know the problem inputs and outputs, develop a list of formulas that specify relationships between them. The general formula:

*total cost =*

*unit cost \* number of units*

computes the total cost of any number of items purchased. Substituting the variables for our particular problem yields the formula:

*total cost of apples =*

*cost per pound \* pounds of apples*

## DESIGN

*Designing the algorithm to solve the problem* requires you to write a step-by step procedure -**the algorithm**- and then verify that the algorithm solves the problem as intended. Writing the algorithm is often the most difficult part of the problem solving process. Don't attempt to solve every detail of the problem initially; instead, discipline yourself to use top-down design. In **top-down design**, you first list the major steps, or subproblems, that need to be solved, then solve the original problem by solving each of its subproblems. Most computer algorithms consist of at least the following subproblems.

**Algorithm for a Program design Problem**

1. Read the data
2. Perform the computations.
3. Display the results.

Once you know the subproblems, you can attack each problem individually.

You may be familiar with top-down design if you use outlines when writing term papers. Your first step is to create an outline of the major topics, which you refine then by dividing each major topic into several subtopics. Once the outline is complete, you begin writing the text for each subtopic.

**Desk checking** is an important part of algorithm design though often. To desk check an algorithm, carefully perform each algorithm step (or its refinements) just as a computer would and verify that the algorithm works as intended. You'll save time and effort if you locate algorithm errors early in the problem solving process.

**IMPLEMENTATION**

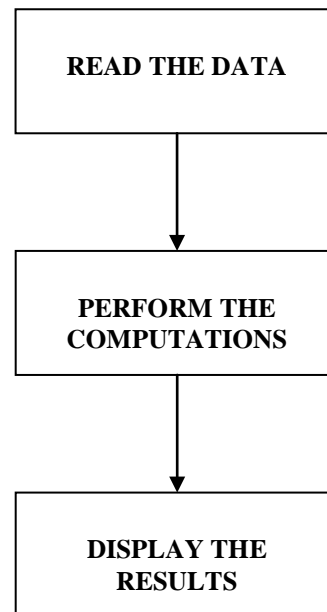
*Implementing the algorithm* involves rewriting the algorithm of a program. You must convert each algorithm step into one or more statements in a programming language.

Structured programming is a disciplined approach to program design that results in programs that are easy to read and understand and less likely to contain errors. The emphasis is on following systematic program style guidelines (which is stressed in this book) to write program code that is clear and readable. Obscure tricks and programming shortcuts are strongly discouraged.

**TESTING**

*Testing and verifying the program* requires testing the completed program to verify that it works as desired. Don't rely on just one test case. Run the program several times using different sets of data, making sure that it works correctly for every situation in the algorithm.

Especially when you test for strange cases, a boundary values in your problem, you will see if your problem design is robust. In the example of the total cost of apples a robust designed program



also returns answers to the questions what of 0 pounds of apples cost.

## MAINTENANCE

*Maintaining and updating the program* involves modifying a program to remove previously undetected errors and to keep it up to date as government regulations or company policy change.

## 4.2 Top-Down Design

Often the algorithm needed to solve a problem is more complex than those we have seen so far and the programmer has to break up the problem into multiple subproblems to develop the program solution. In attempting to solve a subproblem at one level, we introduce new subproblems at lower levels. This process, called **top-down design**, proceeds from the original problem at the top level to the subproblems at each lower level. The splitting of a problem into its related subproblems is analogous to the process of refining an algorithm.

A major ingredient of top-down program design is the **subroutine**. In this section, you will use top-down program design and learn how subroutines can help you subdivide any problem into smaller parts which are easier to manage.

**Note:** A subroutine is a collection of programming statements designed to perform a specific task.

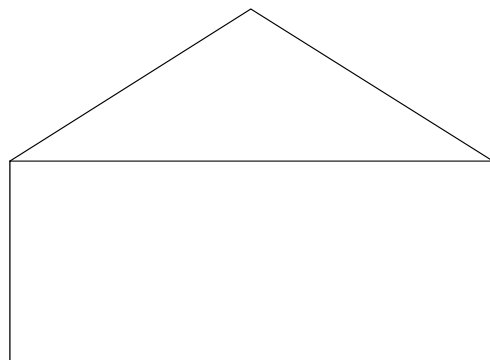
Top-down program design is learned appropriately by applying it to several programs. In order to learn how to use it you need to solve different types of problems.

### 4.2.1 Drawing a House: An example

#### Problem

You want to draw a house on the screen as shown to the right. The user of the final program must be able to choose the position (coordinates) of the house and the house should always be visible on the screen.

To deal with this problem we use the Software Development method described in section 4.1.



## Analysis

The house is formed by a frame (rectangle) with a roof (triangle without its base) on top. Thus we can draw the house using six lines.

When drawing these six lines we have to consider the other problem definitions given at the start:

1. The user must be able to choose the position of the house.
2. The user must be able to choose the size of the house.
3. The house must be completely visible on the screen.
4. Make the drawing of the house.

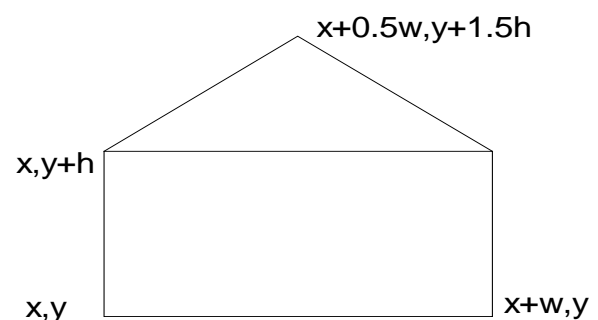
The analysis can now be continued by solving each of the questions above by treating them as separate (sub)problems.

Ad 1. One option to make the user choose the position of the house is by making use of variables. For instance by choosing the left bottom corner as a starting coordinate (x,y) for the house, the position of the house will be fixed for every value of x and y.

Ad 2. If we know the starting point of the house is coordinate (x,y), the width and height can also be set using variables. Let's choose variable w for the width of the house and h for the height of the frame of the house.

The top of the roof we choose as  $0.5 \cdot h$  above the frame thus making the house  $1.5 \cdot h$  high.

The house can then be expressed in variables instead of numbers for all coordinates of the house. See the drawing to the right.



Ad 3. Visualizing the house on the screen for each chosen coordinate is also possible now by adjusting the window settings relative to the variables x,y and w,h.

## Design

A possible **design** of the program is shown to the right. The subproblems defined in the analysis phase are treated in the design as separate routines, which are sequentially solved. Starting with the location (coordinates) of the house to

```
House()
Prgm
" This program draws a house
```

```
Input of location of the house
Input of width and height of the
house
Set window variables
Draw House
```

```
EndPrgm
```

finally drawing the house at the end of the program.

### Implementation

The next step in the software development method is to write small programs (subroutines) to solve each subproblem of the program as defined in the design and analysis phase. To structurize the programming and to be able to identify the separate subproblems. We will use a new programming technique consisting of subroutines.

Subroutines are “small” programs which are incorporated in the program completely.

Subroutines are generally placed in top of the program using the

```
Define subroutine_name()=Prgm
Prgm..
```

```
.....
```

```
.....
```

```
EndPrgm
```

commands.

The subroutine will not be executed until the subroutine name is called for in the main routine.

The main routine is in fact the actual program, which consists primarily of calls for a number of subroutines (subprograms) and perhaps a few extra commands.

An example of a program with subroutines is shown to the right. The command Local is used to let the subroutine exist only within the program when the program is executed. At the end of the program execution the subroutine will be erased from memory. The command Define in conjunction with Prgm...Endprgm declare the subroutine. It will not execute the subprogram. In a later stadium the subprogram can be called or executed by calling the subroutine name in the main routine. Like any program the subroutine must be defined and called using brackets like normal programs. For instance the command subrout1() called in the main routine will execute the subroutine subrout1().

Now we will write our program House().

Start this program in the Program Editor and type the program listing for the location subroutine, making sure to indent for readability. Press Enter at the end of each line.

```
Example()
```

```
Prgm
```

```
" This program is an example of using
subroutines.
```

```
" ***** Subroutines *****
```

```
Local Subrout1
```

```
Define Subrout1()=Prgm
```

```
" Contents of subroutine are
program lines
```

```
...line 1
```

```
...line 2
```

```
...etc.
```

```
EndPrgm
```

```
Local Subrout2
```

```
Define Subrout2()=Prgm
```

```
" Contents of subroutine are
program lines
```

```
...line a
```

```
...line b
```

```
...etc.
```

```
EndPrgm
```

```
" ***** Main Routine *****
```

```
ClrIO
```

```
" Calling the subroutines
```

```
Subrout1()
```

```
Subrout2()
```

```
EndPrgm
```

**Input of the location of the house**

Using the analysis made earlier, it is now easy to think of a way to ask for the location of the house. The subroutine we will call Location()

```
House()
Prgm
" This program draws a house

" Subroutine Location
Local location
Define location()=Prgm
Input "X-coord. of lower left corner",x
Input "Y-coord. of lower left corner",y
EndPrgm
```

**Input of width and height of the house**

Now enter the subroutine as shown on the right in order to ask the user for the width and height of the house.

```
" Subroutine Width and height Local size
Define size()=Prgm
Input "Width of house",w
Input "Height of house",h
EndPrgm
```

**Set window variables**

To place the house in the center of the window you must calculate the exact coordinates of the house and add some extra space on all sides as shown in the subroutine to the right. In this example we choose that the house is centered in the screen keeping a distance of  $0.5*w$  to the left and right and  $0.5*h$  to the top and bottom of the screen.

```
" Subroutine Set window variables
Local Window
Define Window()=Prgm
x-0.5*w»xmin
x+1.5*w»xmax
y-0.5*h»ymin
y+2.0*h»ymax
Endprgm
```

**Draw the house**

Now enter the drawing subroutine as shown to the right. These subroutines follow the window subroutine. As you can see all the line commands are completely described in the variables x,y,w and h

```
" Subroutine Draw house
Local drwhouse
Define drwhouse()=Prgm
Line x,y,x+w,y
Line x,y,x,y+h
Line x,y+h,x+w,y+h
Line x+w,y,x+w,y+h
Line x,y+h,x+0.5*w,y+1.5*h
Line x+w,y+h,x+0.5*w,y+1.5*h
EndPrgm
```

**The Main Routine**

All subproblems are solved and you are now ready to write the main routine in which all subroutines are activated consecutively. Your design helped you do this quickly. Use a lot of comment to describe the meaning of the subroutines.

```
" ***** MAIN ROUTINE *****

Location() " Location subroutine
Size() " Size subroutine
Window() " Settings subroutine
Drwhouse() " Draw house subroutine

EndPrgm
```

The complete house program is shown to the right.

Review the house program. Notice how the comment and indentation help you to follow the flow of the program.

The last EndPrgm statement was placed in the program template when you created the house program.

Think about how the program deals with the values entered by the program user and how these values influence the window settings and position of the house on the screen.

```
House()
Prgm
" This program draws a house

" Subroutine Location
Local location
Define location()=Prgm
Input "X-coord. of lower left corner",x
Input "Y-coord. of lower left corner",y
EndPrgm

" Subroutine Width and height
Local size
Define size()=Prgm
Input "Width of house",w
Input "Height of house",h
EndPrgm

" Subroutine Set window variables
Local Window
Define Window()=Prgm
x-0.5*w»xmin
x+1.5*w»xmax
y-0.5*h»ymin
y+2.0*h»ymax
Endprgm

" Subroutine Draw house
Local drwhouse
Define drwhouse()=Prgm
Line x,y,x+w,y
Line x,y,x,y+h
Line x,y+h,x+w,y+h
Line x+w,y,x+w,y+h
Line x,y+h,x+0.5*w,y+1.5*h
Line x+w,y+h,x+0.5*w,y+1.5*h
EndPrgm

" ***** MAIN ROUTINE *****

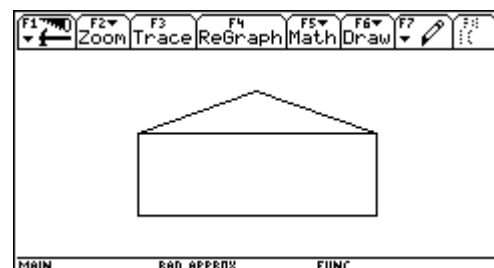
" Calls the subroutines to be executed
Location() " Location subroutine
Size()    " Size subroutine
Window()  " Settings subroutine
Drwhouse() " Draw house subroutine

EndPrgm
```

To run the program return to the Home screen and type House() at the command line. Enter different values for the x, y, w and h variables and interpret the differences between the output (the picture of the house) for these values.

### Testing

In the next the program will be tested. In this phase you will try out the program several times using different sets of values to look for any strange output or errors you did not expect when you wrote the program. If the program does not





work due to an error, you will have to walk through your program steps to locate the error in the program.

## Maintenance

The last step in solving the problem is maintenance. In this phase you will dot your i's and cross your t's to your program. You will try to make the program look good and look the same each time you start the program.

For instance if you run the house program several times you will notice that the input statements which ask the coordinates and size of the house are not erased from the PrgmIO screen.

If you have functions defined in the Y-editor they are also shown in the graph screen. This problem can be solved by adding the ClrIO, ClrGraph, ClrDraw and FnOff statements in the main routine

Furthermore, you will want to get rid of any variables like x,y,h and w created while using the program but which are not necessary outside the program. The variables can be deleted inside the program by using the Delvar variable- name command as last line in the main routine.

The main routine after maintenance is shown on the right.

```

***** MAIN ROUTINE *****

" Clear the IO and graph screen
ClrIO
ClrGraph
ClrDraw
FnOff

" Calls the subroutines to be executed
Location() " Location subroutine
Size()    " Size subroutine
Window()  " Settings subroutine
Drwhouse() " Draw house subroutine

" Clears variables from memory
Delvar x,y,w,h

EndPrgm

```

### 4.3 Summary of commands

**Define** *progName*( ) = **Prgm**

*block of statements*

**EndPrgm**

Creates *progName* as a program or subprogram. *progName*( ) can execute a block of multiple statements. *block* can also include expressions and instructions without restrictions.

**DelVar** *var1* [, *var2*][, *var3*]

Deletes the specified variables from memory.

**Local** *var1* [, *var2*][, *var3*]

Declares the specified vars as local variables. Those variables exist only during evaluation of a program or function and are deleted when the program or function finishes execution.

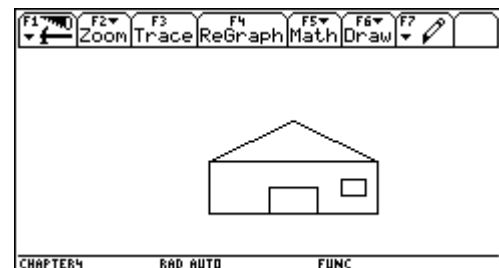
**Note:** Local variables save memory because they only exist temporarily. Also, they do not disturb any existing global variable values. Local variables must be used for temporarily saving values in a multiline function since modifications on global variables are not allowed in a function.

## 4.4 Practical problems

Now try the following exercises. Be sure to use the top-down program design techniques and indentation formatting you learned in this chapter.

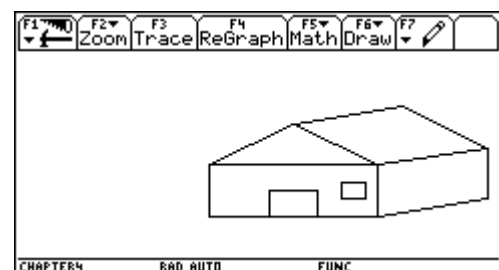
### *Problem 1*

Add a subroutine to draw a window and door in the house program. One possible placement for a window and door is shown to the right.



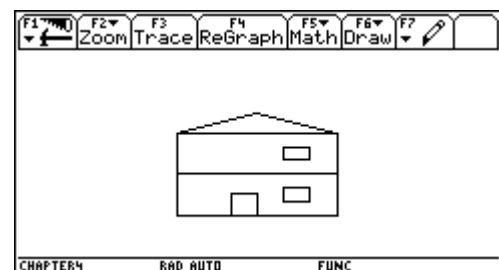
### *Problem 2*

Modify the house program in such a way that the house is drawn "three-dimensional" as shown to the right.



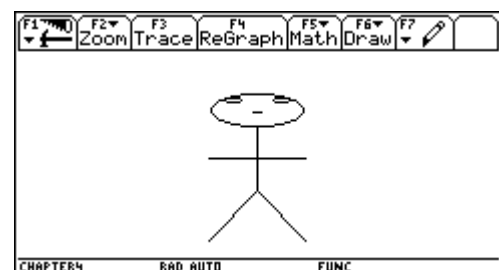
### *Problem 3*

Modify the house program in such a way that the house, with a window and door, becomes a small flat with two floors " as shown to the right.



### *Problem 4*

Use top-down program design to draw a "stick figure" as shown to the right.



### *Problem 5*

Write a program design that can be used to develop a program that draws a picture of a car.

### *Problem 6*

Write a program design that can be used to develop a program that draws a picture of an airplane viewed from the top.

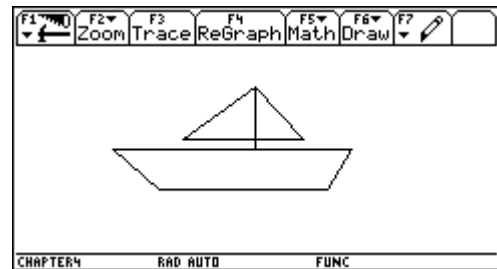
**Problem 7**

The program below draws a sailing ship.

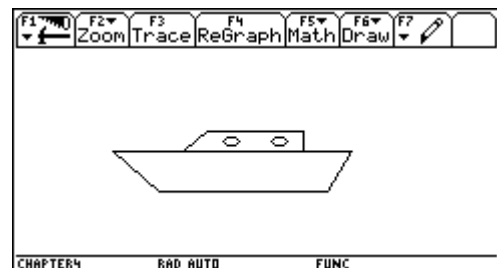
```

Sailboat()
Prgm
  " Draws a sailboat
  Local winset
  " Sets Window Variables
  Define winset()=Prgm
    ClrDraw
    0→xmin
    100→xmax
    0→ymin
    100→ymax
  EndPrgm
  " Draws the Hull
  Local hull
  Define hull()=Prgm
    Line 20,50,70,50
    Line 70,50,65,30
    Line 65,30,30,30
    Line 30,30,20,50
  EndPrgm
  " Draws the Mast
  Local mast
  Define mast()=Prgm
    Line 50,50,50,80
  EndPrgm
  " Draws the Sail
  Local sail
  Define sail()=Prgm
    Line 50,55,35,55
    Line 35,55,50,80
    Line 50,80,60,55
    Line 60,55,50,55
  EndPrgm
  " ***** MAIN ROUTINE *****
winset()
hull()
mast()
sail()
EndPrgm

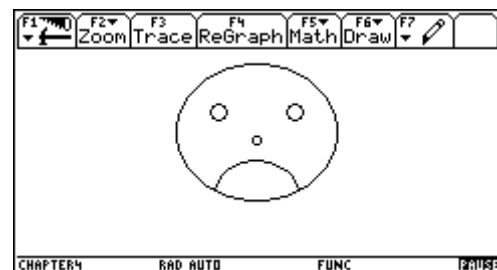
```



Modify the sailboat program in such a way that the boat looks like a cabin cruiser. One possible cabin cruiser is shown to the right.

**Problem 8**

Write a program that draws first a "unhappy face" and next the "happy face" as shown to the right.



## 5. Selection Control Structures

Program design provides control structures; special statements that divert executions from the usual top to bottom sequence.

Some problem solutions require that choices are made between alternative steps, for which some statements may or may not be executed, depending if a certain **condition** (or *Boolean expression*) is **true** or **false**. A **selection control structure** chooses which alternative to execute. The basic condition execution consists of an **If-statement**. The if-statement determines which of several alternatives executes in a particular situation.

The **If-statement** has a number of extended forms that are presented in the next sections.

### 5.1 The If...Then...EndIf statement

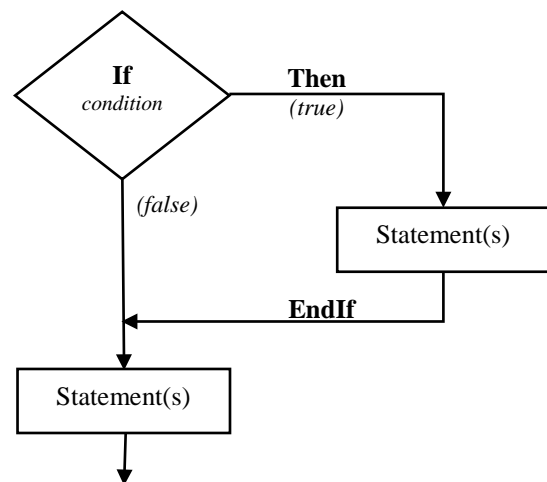
The figure on the right is a flowchart of the **If...Then...EndIf** statement.

A flowchart is a diagram that shows the step by step execution of a control structure (or a program fragment).

The **If condition Then statement(s)** has one alternative; a sequence of one or more *statement(s)*.

#### Boolean expression

A Boolean expression represents a logical **condition**, which is either **true** or **false**.



You can select the **If...EndIf** structures from the Control menu in the Program Editor by pressing **[F2]** and selecting **2:If...EndIf**, or you can type it.

#### 5.1.1 Order a pair of data values: An example

In many program design problems, you must order a pair of data values in memory in such a way that the smaller value is stored in one variable (say, X) and the larger value in another (say, Y).

We can use the **If condition Then statement(s)** to rearranges any two values stored in X and Y in



such a way that the smaller number is in X and the larger number is in Y.

If the two numbers are already in the proper order, the *statement(s)* is not executed.

Although the values of X and Y are being switched, an additional variable, Temp, is needed to store a copy of one of these values.

The table shown on the right simulates the execution of this if statement when X is 45 and Y is 7. Each line shows the part of the if statement that is being executed, followed by its effect. If any variable gets a new value, its new value is shown on that line. If no new value is shown, the variable retains its previous value.

The last value stored in X is 7, and the last value stored in Y is 45.

Now suppose that X is 25 and Y is 8.

Step by step simulation of if statement

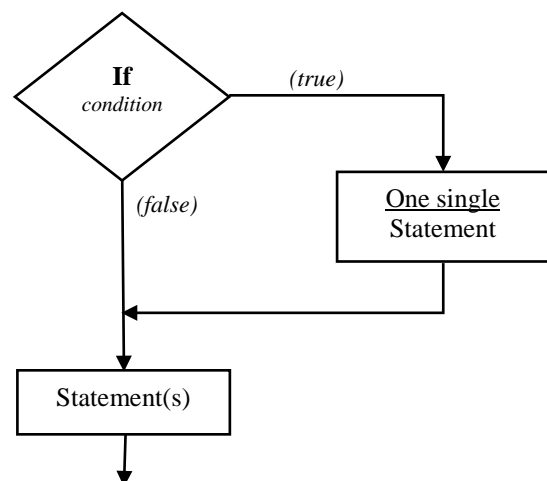
Statement	x	y	temp	Effect
	45	7		
if x>y then				45>7 is <b>true</b>
x→temp			45	stores old x in temp
y→x	7			stores old y in x
temp→y		45		stores temp=old x in y
--EndPrgm				
Statement	x	y	temp	Effect
	25	8		
if x>y then				8>25 is <b>false</b>
	25	8		
--EndPrgm				

1. Start a new program and name it orderxy.
2. Press **ENTER** to store the program name and display the new program template.
3. Type the program lines shown to the right, making sure to indent for readability. Press **ENTER** at the end of each line.
4. Run the orderxy program.

```

orderxy()
Prgm
" Order a pair of data values
ClrIO
Input "Enter the first number:",x
Input "Enter the second number:",y
" Rearrange if necessary
If x>y then
  x→temp " stores old x in temp
  y→x    " stores old y in x
  temp→y " stores temp=old x, in y
EndIf
Disp "The larger value = ", y
Disp "The smaller value = ", x
EndPrgm

```



**Note:** The *If...condition...one single-statement*. When the *condition* is true and you only have to execute one single statement then you may omit the reserved words **"then"** and **"EndIf"**.

### 5.1.2 Is the number an integer? : An example

In mathematics, *integer numbers* are positive or negative whole numbers. An integer contains neither a decimal point nor an exponent. Thus an

integer number is simply a sequence of digits, preceded (optionally) by a sign.

Suppose that you want to find out if a number is an integer or not.

See the program lines as shown to the right.

The function **iPart**(*number*) returns the integer part of the *number*.

If the integer part of *x* is not equal *x* then "not" is displayed and on the next line "integer".

If the integer part of *x* is equal *x* then "not" is not displayed.

```
Integer()
Prgm
" Calculates if number is integer
Input "Enter a number:",x
" Check if "integer part" ≠ x
If iPart(x)≠x
  Disp "not" " one single statement
Disp "an integer"
EndPrgm
```

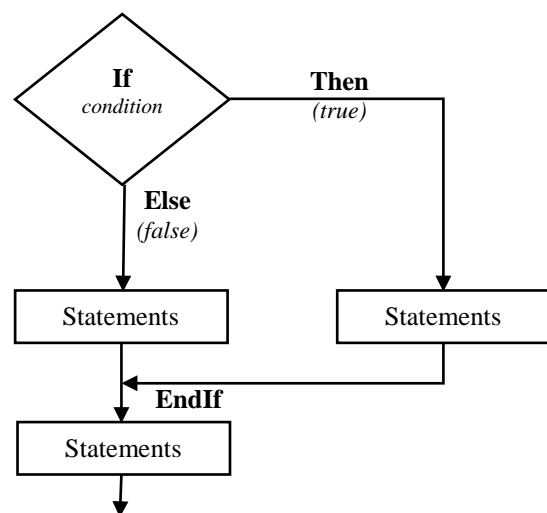
## 5.2 The If...Then...Else...EndIf statement

Unlike the previous example, there are times when you must choose between two different options, rather than simply choosing or ignoring an option.

### 5.2.1 The absolute value of a number: An example

The absolute value of a number *X* is the positive value of *X*, whether it is originally negative or positive. For example +5 stays +5 and -5 becomes +5.

Suppose that you want to define the absolute value function of a random number. This can be done by using the **If...Then...Else...EndIf** statement (*instead of relying on the abs() function available on the TI-92*)



1. Start a new program in the Program Editor and name it absvalue.
2. Enter the program lines as shown to the right. To type the "greater than or equal to" symbol, press **[2nd][>][=]** or **[♦][>]** or press **[2nd][MATH]**, select 8:Test, and select 3:≥.
3. Run the program.

If **x** is greater than or equal to zero, the **Then** statement is executed, and **AbsX** is **x** itself; but if **x** is less than zero, the **Else** statement is executed, and **AbsX** is the opposite of **x**.

```
absvalue()
Prgm
" Calculates the absolute value
Input "Enter a number:",x
" Check if x ≥ 0
If x≥0 Then
  x→AbsX
Else
  -x→AbsX
EndIf
Disp "Absolute value of",x,"is",AbsX
EndPrgm
```

### 5.2.2 Rental Car Pricing: An example

Suppose you are interested in renting a car for the summer, and you want to choose between an economy car at \$20 a day or a midsize car at \$30 a day. You want to compare the costs of the two cars if you're renting them for a specific number of day's (N).

You can use the If...Then ...Else...EndIf structure to solve this problem.

Start a new program in the Program Editor and name it **rentcar1**. Assume that:

Choice=1 for an economy car and

Choice=2 for a midsize car,

and enter the program lines as shown to the right.

```
rentcar1()
Prgm
" Rental Car Pricing
" Compute the Rental Cost
" Choose type of car
Disp "Enter car choice"
Disp "1=Economy, 2=Midsize"
Input "Give Choice:",choice
Input "Enter Renting days",n
If choice=1 Then
  20*n→cost
Else
  30*n→cost
EndIf
Disp "The total cost is:",cost
EndPrgm
```

### 5.3 The If...Then...ElseIf...EndIf statement

Let's look at the **rentcar1** example.

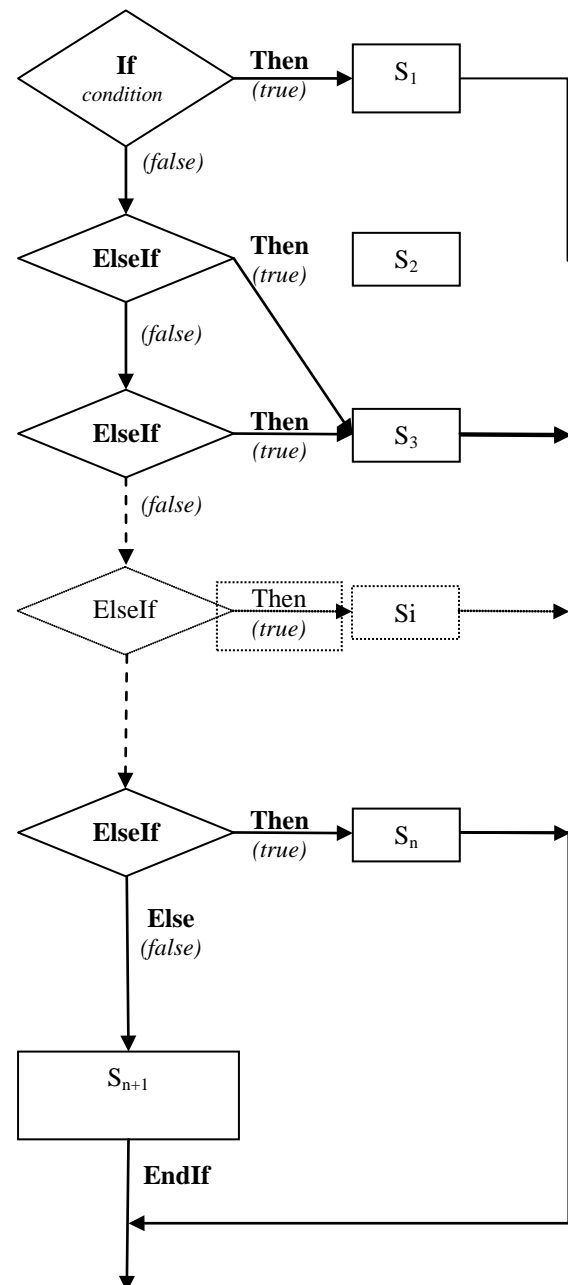
What if Choice is equal to a number other than 1 or 2, for example 3 or 1.2?

The statements above would still calculate a cost of \$30 a day!

An extension of the If ... Then ... Else ... EndIf structure, the **If...Then...ElseIf...EndIf** statement will take care of these difficulties and give you as many options as you need.

Look at the flowdiagram shown on the right to see if you can figure out how to correct the problem.

Now modify the rentcar program as shown on the next page.





What's happening here?

If Choice=1, the cost is figured at \$20 a day.

If Choice $\neq$ 1, the program goes to the second condition to see if Choice=2; if so, the cost is equal to \$30 a day.

If Choice $\neq$ 1 AND Choice  $\neq$ 2, the program goes to the third condition to see if Choice=3; if so, the cost is equal to \$40 a day.

If Choice  $\neq$ 1 AND Choice  $\neq$ 2 AND Choice  $\neq$ 3 AND Choice  $\neq$ 4, the program goes to the Else statement and assigns an negative value -1 to Cost.

Knowing that Cost is always positive you can use the negative value of Cost as an "error statement".

```
rentcar2()
Prgm
  " Rental Car Pricing
  " Compute the Rental Cost
  " Choose type of car
  Disp "Enter car choice"
  Disp "1=Economy, 2=Midsize"
  Disp "3=Full size, 4=Sports car"
  Input "Give Choice:",choice
  Input "Enter Renting days",n

  " Check for the correct choice
  If choice=1 Then
    20*n $\rightarrow$ cost
  ElseIf choice=2 Then
    30*n $\rightarrow$ cost
  ElseIf choice=3 Then
    40*n $\rightarrow$ cost
  ElseIf choice=4 Then
    50*n $\rightarrow$ cost
  Else
    " invalid choice, cost always  $\geq$  0
    -1 $\rightarrow$ cost
  EndIf

  If cost $\geq$ 0 Then
    Disp "the total cost is:",cost
  Else
    Disp "Invalid choice"
  End
EndPrgm
```

## 5.4 Top-Down Program Design

Now, let's discuss how you could change the statements in the rentcar program to use the top-down program design and block structure ideas from chapter 4.

You may wonder why you would want to do this with such a simple program. After all, this problem is easy to program directly without a top-down approach as shown in rentcar2(). Resist the temptation to skip the top-down approach even in the easiest program design problems. The thinking process you will develop can pay big dividends when you work on more challenging problems. By practicing these structured programming skills, you will become adept at solving complex problems more easily.

## Completing the Rental Car Program

A program is made up of three parts:

1. an input block,
2. an action (calculation) block, and
3. an output block.

You could represent these three tasks for this program in a design as shown on the right. By breaking down the problem into smaller parts, you can focus on each task separately.

Recall from Chapter 4 that the commands **Local** and **Define** are necessary in defining a subroutine.

1. Enter the **indata** subroutine as shown to the right at the beginning of the program after the "Rental Car Pricing" comment.
2. Next, convert the If...Then...ElseIf...EndIf structure into the **rentcost** subroutine by placing Local rentcost and Define rentcost()=Prgm before the structure and **rentcost()**=Prgm before the structure and **EndPrgm** after it.
3. Finally, create the output subroutine **outdata** by including the If...Then...Else statements that display the results. This subroutine follows the rentcost subroutine.
4. Complete the program by adding the main routine statements to call the subroutines.
5. Return to the Home screen and run the rentmain() program. Be sure to check the various choices, including at least several one invalid choices like 6, 11.2 or -5.

```
rentmain()
Prgm
" Rental Car Pricing
  Input Subroutine
  Compute Rental Cost Subroutine
  Output Subroutine
" ***** MAIN ROUTINE *****
  Invoke Subroutines
EndPrgm
```

```
" Choose type of car
Local indata
Define indata()=Prgm
  Disp "Enter car choice"
  Disp "1=Economy, 2=Midsize"
  Disp "3=Full size, 4=Sports car"
  Input "Give Choice:",choice
  Input "Enter Renting days",n
EndPrgm

" Compute the Rental Cost
Local rentcost
Define rentcost()=Prgm
  If choice=1 Then
    20*n>cost
  ElseIf choice=2 Then
    30*n>cost
  ElseIf choice=3 Then
    40*n>cost
  ElseIf choice=4 Then
    50*n>cost
  Else
    -1>cost
  EndIf
EndPrgm

" Display the Total Cost
Local outdata
Define outdata()=Prgm
  If cost≥0 Then
    Disp "the total cost is:",cost
  Else
    Disp "Invalid choice"
  EndIf
EndPrgm
```

```
rentmain()
Prgm
" Rental Car Pricing
-----
-----
-----
" ***** MAIN ROUTINE *****
ClrIO
ClrHome
indata()
rentcost()
outdata()
EndPrgm
```

### 5.4.1 The ABC Formula: An example

#### Problem

Let's try an example from algebra: the solution to a quadratic equation:

$$ax^2 + bx + c = 0$$

for x.

#### Analysis

The quadratic formula, also called the ABC formula:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

solves the above equation.

The problem can yield three different solutions depending the values of a, b, and c. In specific the value of the root section of the solution called the discriminant has an important influence.

1) Discriminant  $> 0$

if  $b^2 - 4ac > 0$  the solution of the quadratic equation has two different real solutions.

2) Discriminant  $= 0$

if  $b^2 - 4ac = 0$  the solution of the quadratic equation has only one real solution.

3) Discriminant  $< 0$

if  $b^2 - 4ac < 0$  no real solutions exist, as the square root of a negative number is not a real number. The solutions will be "complex numbers".

To see which solution the quadratic equation will yield a precalculation of the discriminant is necessary after which the correct solution can be displayed on the screen.

#### Design

The basic structure can be split into three basic "subroutines" or "subprograms" as shown on the right.

#### complex numbers

If you try  $\sqrt{-1}$  on your TI-92 then the response is: "Non-real result".

$\sqrt{-1}$  is an example of a special set of numbers called **complex numbers**. Because of the importance of complex numbers in engineering and math a special symbol, **i**, is reserved for  $\sqrt{-1}$ .

We write  $i \equiv \sqrt{-1}$

You will find this notation on the TI-92 using  $\boxed{2nd} \boxed{[i]}$

Knowing this, roots of negative values can still be determined.

For example  $\sqrt{-4} = \sqrt{4} * \sqrt{-1} = 2i$

```

abcform1()
Prgm
  " Solutions of Quadratic Equation

  " ***** SUB ROUTINES *****
  Enter Coefficients Subroutine
  Calculate Solutions Subroutine
  Display Solutions Subroutine

  " ***** MAIN ROUTINE *****
  Invoke Subroutines
  EndPrgm

```

## Implementation

The first subroutine Enterdat() is shown to the right.

Note : To type  $2^2$  Press  $[2^{nd}][Char]:Math\rightarrow 2^2$

The second subroutine Calcsolu() calculates the solution of the quadratic equation for a given a, b and c. It first calculates the first part of the solution and the discriminant. Depending the value of the discriminant (Disc) two possible values of the second part of the solution are available:

If  $Disc < 0$  then  $part2 = \frac{\sqrt{Disc}}{2 \cdot a}$  or,

If  $Disc < 0$  then  $part2 = \frac{\sqrt{Disc}}{2 \cdot a} \cdot i$  as complex

solution.

The use of the If...Elseif...EndIf-statement can be used to implement these solutions in the program.

The third and last subroutine Dispsol1() displays the solutions of the equation on the screen. As with the calculation subroutine multiple solutions are possible depending the value of the discriminant. A possible solution of this subroutine with the use of the If...Elseif...Else...EndIf-statement is shown on the right.

Finally the main routine can be added to complete the program.

```

" Enter coefficient Subroutine
Local enterdat
Define enterdat()=Prgm
Disp "Compute Solutions of:"
Disp " ax2 + bx + c = 0"
" Enter coefficients
Input "Enter a:",a
Input "Enter b:",b
Input "Enter c:",c
EndPrgm

```

```

" Calculation Subroutine
Local calcsolu
Define calcsolu()=Prgm
a2b/(2*a)»part1
b2-4*a*c»disc
If disc>0 Then
§(disc)/(2*a)»part2
Elseif disc<0 then
§(a2disc)/(2*a)*(i)»part2
EndIf
EndPrgm

```

```

" Display solution option 1
Local dispsol1
Define dispsol1()=Prgm
If disc>0 Then
Disp "The real solutions are:"
Disp "x1=",part1,"+",part2
Disp "x2=",part1,"-",part2
Elseif disc<0 Then
Disp "The complex Solutions are:"
Disp "x1=",part1,"+",part2
Disp "x2=",part1,"-",part2
Else
Disp "The only real solution is:"
Disp "x=",part1
EndIf
EndPrgm

```

```

" ***** MAIN ROUTINE *****

```

```

" Calling subroutines
enterdat()
calcsolu()
dispsol1()

```

```

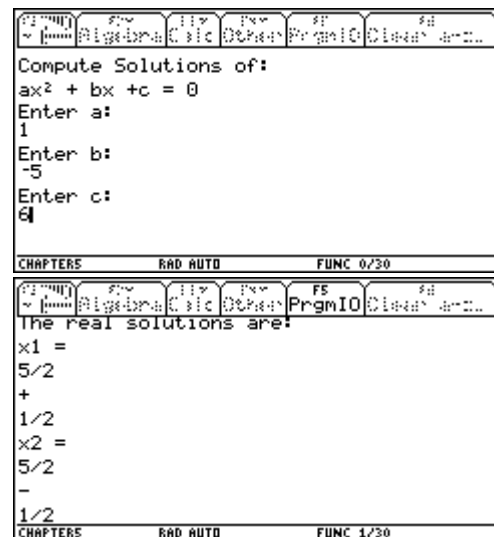
EndPrgm

```

The program can now be tested to check if the program works correctly.

A possible input and solution are given on the right.

If you run this program every expression in the Disp statement is printed out on a separate line as shown on the right.. In other words, the text "x1=" or "x2=" is printed on one line, the number Part1 on a second line, the symbol "+" or "-" on a third line, and the number Part2 on a fourth line.



You can get around this difficulty by converting the numbers and symbols into a string. Disp applied to a string prints out the string as a single object on one line. Applying the string command to a numerical expression changes it into a string. The & symbol concatenates two strings to one string.

Your original solution then becomes  
 "x1 = "&string(Part1)&" +(" &string(Part2),  
 or we can first add Part1+Part2 and then use the string command: "x1 = "&string(Part1 +Part2)  
 An example is shown on the right.

Substituting this new version for the output into the subroutine gives you the revised **dispsol2()** subroutine shown on the right.

To type &, press [2<sup>nd</sup>]H.

Notice that the last condition Elseif Disc<0 is not necessary and could be replaced by an Else since once Disc>0 and Disc=0 have been dealt with, Disc<0 is the only possibility remaining (unlike the rental car situation, where other possibilities remained).

### Maintenance

To finish the program you can modify the main routine to clear the PrgmIO and Home screen. You can also add a command to delete all used variables.

```

" Display Solution option 2
Local dispsol2
Define dispsol2()=Prgm
  If disc>0 Then
    Disp "The real solutions are:"
    Disp "x1 = "&string(part1+part2)
    Disp "x2 = "&string(part1-part2)
  Elseif disc=0 Then
    Disp "The only real solution is:"
    Disp "x = "&string(part1)
  Else
    Disp "The complex solutions are:"
    Disp "x1 = "&string(part1+part2)
    Disp "x2 = "&string(part1-part2)
  Endif
EndPrgm

```

```

" ***** MAIN ROUTINE *****
" Cleaning screens
ClrIO
ClrHome

" Calling subroutines
enterdat()
calcsolu()
dispsol2()

" Deleting variables
DelVar part1,part2,disc,a,b,c

EndPrgm

```



Run the abcform() program. Check it by entering values for a, b, and c that produce positive, zero, and negative discriminants.

*You also may want to check your results by using the solve() or csolve() functions, which are built into the TI-92.*

```

abcform()
Prgm
" Solution of quadratic Equation

" **** Subroutines ****
" Enter coefficient Subroutine
Local enterdat
Define enterdat()=Prgm
Disp "Compute Solutions of:"
" Enter coefficients
Input "Enter a: ",a
Input "Enter b: ",b
Input "Enter c: ",c
EndPrgm

" Calculation Subroutine
Local calcsolu
Define calcsolu()=Prgm

$$b/(2*a) \rightarrow \text{part1}$$


$$b^2-4*a*c \rightarrow \text{disc}$$

If disc>0 Then
  
$$\sqrt{\text{disc}}/(2*a) \rightarrow \text{part2}$$

Elseif disc<0 Then
  
$$\sqrt{\text{disc}}/(2*a) \rightarrow \text{part2}$$

EndIf
EndPrgm

" Display Solution
Local dispsolu
Define dispsolu()=Prgm
If disc>0 Then
  Disp "The real solutions are:"
  Disp "x1 = "&string(part1+part2)
  Disp "x2 = "&string(part1-part2)
Elseif disc=0 Then
  Disp "The only real solution is:"
  Disp "x = "&string(part1)
Else
  Disp "The complex solutions are:"
  Disp "x1 = "&string(part1+part2)
  Disp "x2 = "&string(part1-part2)
EndIf
EndPrgm

" ***** MAIN ROUTINE *****
" Cleaning screens
ClrIO
ClrHome

" Calling subroutines
enterdat()
calcsolu()
dispsolu()

" Deleting variables
DelVar part1,part2,disc,a,b,c

```

### 5.4.2 Assigning Exam Scores to Letter Grades: An example

Let's look at one last example. Suppose you want to assign letter grades based on exam scores.

Exam Score	Letter Grade Assigned
90 and above	A
80 to 89	B
70 to 79	C
60 to 69	D
below 60	F

You realize that entering the score is a single program line that does not require a subroutine. You then write your program design.

First you can write the subroutine to determine Letter Grades from Exam score using an If...Then...ElseIf...EndIf structure. Second to display the assigned letter grade.

1. Start a new program and name it gradass1
2. Enter the compgrad subroutine at the beginning of the program.  
Note: The case of an invalid score being entered is considered. It is always good to let the user know when inappropriate data has been entered.

```
gradass1()
Prgm
" Convert Point Score to Letter Grade
Determine Letter Grade Subroutine
Output Grade Subroutine
" MAIN ROUTINE
Input the Total Point Score
Invoke the Subroutines
EndPrgm
```

```
gradass1()
Prgm
" Convert Point Score to Letter Grade
" Determines Letter Grade from Point
Score

Local compgrad
Define compgrad()=Prgm
If score ≥90 Then
  "A"→grade
Elseif score<90 and score≥80 Then
  "B"→grade
Elseif score<80 and score≥70 Then
  "C"→grade
Elseif score<70 and score≥60 Then
  "D"→grade
Elseif score<60 and score≥0 Then
  "F"→grade
Else
  Disp "Invalid Score."
EndIf
EndPrgm
```



3. You are now ready to complete the program `gradass1()` as shown to the right. The `dispgrad` subroutine displays the letter grade.
4. Run the program. Check the output with the table showing the Total Score and its corresponding Letter Grade to make sure the program works correctly. It is also important to test invalid scores.

```

gradass1()
Prgm
" Convert Point Score to Letter Grade
" Determines Letter Grade from Point
  Score

" ***** SUBROUTINES *****
Local compgrad
Define compgrad()=Prgm
  If score≥90 Then
    "A"→grade
  Elself score<90 and score≥80 Then
    "B"→grade
  Elself score<80 and score≥70 Then
    "C"→grade
  Elself score<70 and score≥60 Then
    "D"→grade
  Elself score<60 and score≥0 Then
    "F"→grade
  Else
    Disp "Invalid Score."
  EndIf
EndPrgm

" Displays the Letter Grade
Local dispgrad
Define dispgrad()=Prgm
  If score≥0
    Disp "The letter grade is:",grade EndPrgm

" ***** MAIN ROUTINE *****
ClrIO
Input "Enter the point score:",score
compgrad()
dispgrad()
EndPrgm

```

## 5.5 Summary of Commands

$^2$	exponent $^2$ To type exponent $^2$ Press $\boxed{2nd}\boxed{[CHAR]2:Math}\blacktriangleright I:2$
$\geq$	"greater than or equal to" symbol To type $\geq$ press $\boxed{2nd}\boxed{[>]}\boxed{=}$ or $\boxed{\blacklozenge}\boxed{.}$ or press $\boxed{2nd}\boxed{[MATH]}$ , select 8:Test, and select 3: $\geq$ .
$\leq$	"smaller than or equal to" symbol To type $\leq$ press $\boxed{2nd}\boxed{[<]}\boxed{=}$ or $\boxed{\blacklozenge}\boxed{0}$ or press $\boxed{2nd}\boxed{[MATH]}$ , select 8:Test, and select 4: $\leq$ .
$\neq$	"not equal" symbol To type $\neq$ press $\boxed{\div}\boxed{=}$ or $\boxed{2nd}\boxed{[V]}$ or press $\boxed{2nd}\boxed{[MATH]}$ , select 8:Test, and select 6: $\neq$ .
<b>&amp;</b>	press $\boxed{2nd}\boxed{H}$ string1 <b>&amp;</b> string2 $\Rightarrow$ string The <b>&amp;</b> symbol stands for appends (concatenates) two strings into one string. <u>For example:</u> "Ton" & "and" & "Martijn" $\boxed{ENTER}$ returns: "Ton and Martijn"
<b>and</b>	Boolean expression1 <b>and</b> Boolean expression2 $\Rightarrow$ Boolean expression Returns true only if both expressions simplify to true. Returns false if either or both expressions evaluate to false. Returns true or false or a simplified form of the original entry. <u>For example:</u> true <b>and</b> true $\boxed{ENTER}$ returns: true false <b>and</b> true $\boxed{ENTER}$ returns: false false <b>and</b> false $\boxed{ENTER}$ returns: false $x \geq 6$ <b>and</b> $x \geq 7$ $\boxed{ENTER}$ returns: $x \geq 7$
<b>not</b> (Boolean expression1)	$\Rightarrow$ Boolean expression Returns true, false, or a simplified Boolean expression1. <u>For example:</u> not( $2 >= 3$ ) $\boxed{ENTER}$ returns: true not( $x < 7$ ) $\boxed{ENTER}$ returns: $x \geq 7$ not(not(trueborn)) $\boxed{ENTER}$ returns: trueborn
<b>or</b>	Boolean expression1 <b>or</b> Boolean expression2 $\Rightarrow$ Boolean expression Returns true if either or both expressions simplify to true. Returns false only if both expressions evaluate to false. Returns true or false or a simplified form of the original entry. <u>For example:</u> true <b>or</b> true $\boxed{ENTER}$ returns: true false <b>or</b> true $\boxed{ENTER}$ returns: true false <b>or</b> false $\boxed{ENTER}$ returns: false $x \geq 7$ <b>or</b> $x \geq 8$ $\boxed{ENTER}$ returns: $x \geq 7$
<b>If expression</b>	If <i>expression</i> is true, only the statement following is executed; otherwise, the statement is skipped.

**If** *expression* **Then**  
    *block of statements*  
**EndIf**

If *expression* is true, the statements in *block of statements* are executed; otherwise, these statements are not executed.

**If** *expression* **Then**  
    *block of statements1*  
**Else**  
    *block of statements2*  
**EndIf**

If *expression* is true, the statements in *block of statements1* are executed; otherwise, the statements in *block of statements2* are executed.

**If** *expression1* **Then**  
    *block of statements1*  
**ElseIf** *expression2* **Then**  
    *block of statements2*  
.  
.  
.  
**ElseIf** *expressionN* **Then**  
    *block of statementsN*  
**EndIf**

If *expression1* is true, the statements in *block of statements1* are executed; otherwise, if *expression2* is true, the statements in *block of statements2* are executed; and so on.

## 5.6 Practical problems

Try the following exercises. Be sure to write a complete program using top-down program design, subroutines, and decision blocks.

### ***Problem 1***

Write a program to roll a pair of dice. Use a random number generator to generate the six possible values of each dice and display "You Win!" if a 7 or 11 is rolled, "Snake Eyes!" if a 2 is rolled and "You loose" otherwise.

Hint:

The TI-92 random-number generator **rand()**. **rand(6)** returns a random integer in the interval [1,6]

### ***Problem 2a***

Write an interactive program that will determine and print the day of the week for a given daynumber dn;  $1 \leq dn \leq 7$  dn=1 correspond to Sunday, 2 to Monday, etc. Display an error for an incorrect value of dn.

### ***Problem 2b***

Write a interactive program that determines the day number (1 to 366) in a year for a date that is provided as input data. in the form mm-dd-yyyy. As an example, January 1, 1994, is day 1; December 31, 1993 (input: 12-31-1993), is day 365 and December 31, 1996 is day 366 since 1996 is a leap year. A year is a leap year if it is divisible by 4, except that any year divisible by 100 is a leap year only if it is divisible by 400.

### ***Problem 2c***

Write an interactive program that will determine and print the day of the week, Sunday, Monday, etc. for a given date, entered in the form mm-dd-yyyy (for example: 12-23-1952:Wednesday). A year is a leap year if it is divisible by 4, except that any year divisible by 100 is a leap year only if it is divisible by 400.

Hint:

You can make use of the following functions on the TI-92: **intDiv()** and **Mod()**.

### ***Problem 3***

Write an interactive program that will ask for 3 integers and determine and print how many has the same value.

### ***Problem 4***

Write an interactive program that sorts 4 user-determined integers.

### ***Problem 5***

Write an interactive program that calculates the total amount due (excluding taxes) when given the number of items purchased and the price per item.

In addition, a 10% discount is given when a customer purchases more than 10 of the same item.

Think about how you might solve the problem before you writing the program.

Hint:

To type the % symbol, press **[2nd][CHAR]**, select 3:Punctuation, and then select 5:%

**Problem 6**

In 1994, Louisiana's state income tax rate was 2% of the first \$10,000 of taxable income, 3% of the next \$10,000 of taxable income, and then increased by 1% for each additional \$10,000 of taxable income, with a maximum rate of 6% of any taxable income above \$40,000. Write a program with an **If...Then...ElseIf...EndIf** structure that asks the user for the taxable income amount and then prints the appropriate Louisiana state income tax.

**Problem 7a**

Write a program that requests the user to select a number from 1 to 5. The program then tells whether the number is prime, composite, or neither. Be sure to deal with the possibility that the user enters an invalid number.

Note: A prime number has exactly two factors, a composite number has more than two factors, and 1 is neither prime nor composite.

**Problem 7b**

Write a program that requests the user to select a number. The program then tells whether the number is prime, composite, or neither. Be sure to deal with the possibility that the user enters an invalid number.

**Problem 8**

Modify the grades program in section 5.4.2 to request each student's name and ID number. Then have the program display the student's name and ID number along with his/her letter grade.

**Problem 9**

Write a program that takes a person's age as input and then prints a description using these categories:

- Infant (under 2)
- Toddler (2 to 3)
- Child (under 13)
- Teenager (13 to 19)
- Adult (18 or older)
- Senior citizen (65 or older).

Hint: You have to be careful here since some of the categories overlap (some ages may have more than one description).

**Problem 10**

Explain why the results of gradass2(), on the right and gradass1() on page 49 are equal?

```
gradass2()
Prgm
  " Convert Point Score to Letter Grade
  " Determines Letter Grade from Point
  Score

  " ***** SUBROUTINES *****
  Local compgrad
  Define compgrad()=Prgm
    If score≥90 Then
      "A">grade
    ElseIf score≥80 Then
      "B">grade
    ElseIf score≥70 Then
      "C">grade
    ElseIf score≥60 Then
      "D">grade
    ElseIf score≥0 Then
      "F">grade
    Else
      Disp "Invalid Score."
    EndIf
  EndPrgm

  " Displays the Letter Grade
  Local dispgrad
  Define dispgrad()=Prgm
    If score≥0
      Disp "The letter grade is:",grade EndPrgm

  " ***** MAIN ROUTINE *****
  ClrIO
  Input "Enter the point score:",score
  compgrad()
  dispgrad()
EndPrgm
```



## 6. Repetition Control Structures

In the programs studied so far, the statements execute only once. Many tasks however are repetitive like for instance the multiplication table of seven as shown on the right. In order to perform this multiplication you have to write the line `Disp "1 x 7 = "&string(1*7)` etc. 9 times. In most software however, a process (or block of statements) can be repeated quite easily many times with a repetition control structures. In the remainder of this book repetition control structures will be called “loops”.

Three different types of loops exist in the TI-92 namely **For...EndFor**, **Loop..EndLoop** and **While...EndWhile**. We will discuss all three types and their advantages of each in the following paragraphs.

### 6.1 The For...EndFor structure

Our study of the loop structures starts with the **For...EndFor** structure.

We can use the **For...EndFor** structure in the multiplication table of 7. Start a new program and name it `table_7b`.

Type the program lines as shown on the right. To create the **For...EndFor** structure select the **4:For...EndFor** command from the Control menu ( `F2` ).

This program has the same output as program `Table_7a` but is much smaller in size. The **For...EndFor** structure performs all repetitions in one small block.

The statement `For n,1,9 ... EndFor` loops the command

**`Disp string(n)&" x 7 = "&string(n*7)`**

9 times, starting with `n=1`, `n=2`, `n=3` ... and ending with `n=9`.

The variable `n` is called a “counter” and by default increments automatically by 1 after each loop repetition. When `n>9` the program exits the **For...EndFor** loop.

It is imminent to know beforehand how many times you want the loop to be executed. The **For...EndFor** structure requires you to know the end value of the counter.

```

tabel_7a()
Prgm
" The multiplication table of 7:
" The "old-fashioned way"
Local oldtbl7a
Define oldtbl7()=Prgm
  Disp " 1 x 7 = "&string(1*7)
  Disp " 2 x 7 = "&string(2*7)
  Disp " 3 x 7 = "&string(3*7)
  Disp " 4 x 7 = "&string(4*7)
  Disp " 5 x 7 = "&string(5*7)
  Disp " 6 x 7 = "&string(6*7)
  Disp " 7 x 7 = "&string(7*7)
  Disp " 8 x 7 = "&string(8*7)
  Disp " 9 x 7 = "&string(9*7)
EndPrgm
" ***** MAIN ROUTINE *****
ClrIO
oldtbl7a()
EndPrgm

```

```

table_7b()
Prgm
" The multiplication table of 7:
" The new "do this 9 times" way
Local newtbl7b
Define newtbl7b()=Prgm
  For n,1,9
    Disp string(n)&" x 7 = "&string(n*7)
  EndFor
EndPrgm
" ***** MAIN ROUTINE *****
ClrIO
newtbl7b()

```

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100

Summarizing:

The **For...EndFor** structure is defined as follows:

For Countvar,Startval,Endval[,step]

Block of Statement(s)

EndFor

Note: Step(size) is the value by which the countervar(iable) increments and can be positive or negative. By default stepsize is 1.

## 6.2 The Loop...EndLoop structure

Suppose you want to roll a set of dice until both dice show the number six on top. In this case you do not know in advance how many times you have to roll the dice. The **For...Endfor** structure cannot be used.

You have to use a loop structure, which loops continuously until a special condition is met to stop the loop. The **Loop...EndLoop** structure can perform this task. Start a new program and call it loopdice. Enter the program lines as shown on the right and run the program.

Pay special attention to how the program exits the loop and how the number of times the loop is executed is counted.

The condition to end the loop structure is placed inside the loop using an **If...Then** statement and the command **Exit**. The flowchart to the right illustrates the execution.

Summarizing:

The Loop...EndLoop structure is defined as follows:

Loop

Block of Statement(s)

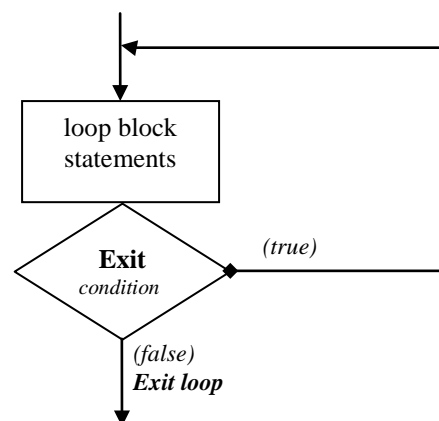
If end\_condition then

Exit

EndIf

EndLoop

```
loopdice()
Prgm
" Count number of rolls until the pair
" {6 6} appear with Random dice
Local dice1,dice2,count
0→count " set count initial to 0
0→dice1 " set dice1 initial to 0
0→dice2 " set dice2 initial to 0
" Creates a Loop
Loop
  count+1→count " increment count
  Rand(6)→dice1
  Rand(6)→dice2
  If dice1=6 and dice2=6 Then
    Exit
  EndIf
EndLoop
Disp "the number of rolls to"
Disp "get {6 6} is "&string(count)
EndPrgm
```





### 6.3 The While...EndWhile structure

Another structure to create loops is the **While...EndWhile** structure. Like the **Loop...EndLoop** structure the **While...EndWhile** can also exit the loop at any given time depending on the exit condition. With respect to the end condition however the **While...EndWhile** structure requires the end condition to be stated at the beginning of the loop instead of inside the loop for the **Loop...EndLoop** structure. This requires you to define in advance all variables used for the end condition.

The program Loopdice can also be made to work with the **While...EndWhile** structure. Make a new program and call it Whildice. Enter the program lines as shown on the right and run the program.

The Boolean expression after the reserved word **While** ( $\text{dice1} \neq 6$  or  $\text{dice2} \neq 6$ ) is the condition to continue the loop. This condition is evaluated every time at the start of the loop. While the expression is true the loop block statements between the **While...EndWhile** statements is repeated. When the expression is false (i.e.  $\text{dice1}=6$  and  $\text{dice2}=6$ ) the loop is exited. This illustrates the flowchart to the right.

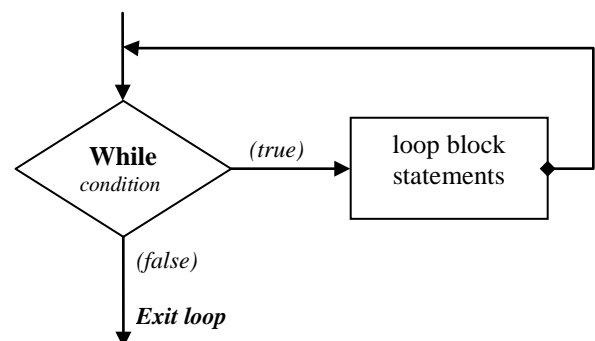
Compare this flowchart with the the **Loop...EndLoop** flowchart and note the difference.

#### Summarizing:

The **While...EndWhile** structure is defined as follows:

While expression is true  
  
Block of Statement(s)  
  
EndWhile

```
whildice()
Prgm
" Count number of rolls until the pair
" {6 6} appear with Random dice
Local dice1,dice2,count
0→count " set count initial to 0
0→dice1 " set dice1 initial to 0
0→dice2 " set dice2 initial to 0
" Creates a Loop
While (dice1 ≠ 6 or dice2 ≠ 6)
  count+1→count " increment count
  Rand(6)→dice1
  Rand(6)→dice2
EndWhile
Disp "the number of rolls to"
Disp "get {6 6} is "&string(count)
EndPrgm
```



## 6.4 Using Loops to Accumulate a Sum and Average

Loops often accumulate a sum of numbers by repeating an addition operation. The average of the set of values can then be found by dividing the number of values in the set.

You can find the average using any of the three repetition structures

On the program design to the right you see that this is the standard input-compute-output problem.

### 6.4.1 Using a For...EndFor Loop

If you know that you have fixed value of N numbers to average you can use a For ... EndFor loop.

Let's concentrate on the subroutine to compute the average. You must request the values from the user, add them up, and then divide by the number of values. The number of values N is in the subroutine to input the numbers, so you can use that number. The first statement of the For ... EndFor loop (For i,1,N) runs the block it controls N times, starting with i=1 and ending when i=N.

You have no way of storing the N values the user enters. If you consider the way averages are calculated, however, you'll realize that you don't need all N individual values, only their sum. Therefore, define a variable Total to be the sum of the previous values. At the beginning  $0 \rightarrow \text{Total}$ , because no values are entered.  $\text{NextX} + \text{Total} \rightarrow \text{Total}$  updates the sum by adding the current value "NextX" to the previous sum "Total" and storing it in the same variable "Total". At the end of the For ... EndFor loop, you'll have the sum of all the values stored in the variable Total.

Start a new program and name it fore\_ave. Enter the inp\_num, calcavg and the out\_avg subroutine as shown to the right. Complete the program by adding the main routine and run the program.

```
average()
Prgm
© Averaging a Set of Numbers
Input Numbers Subroutine
Compute Sum and Average Subroutine
Output the Average Subroutine
© MAIN ROUTINE
Invoke Subroutines
EndPrgm
```

```
fore_ave()
Prgm
" Calculate Avg with For EndFor loop

" Input Number
Local inp_num
Define inp_num()=Prgm
Input "Averaging how many values?",n
EndPrgm

" Calculate Average
Local calcavg
Define calcavg()=Prgm
" initialize variable total
O→total
Disp "Enter the values one at a time"
For i,1,n
Input "Enter next value:",nextx
nextx+total→total
EndFor
total/n→avg
EndPrgm

" Output Average
Local out_avg
Define out_avg()=Prgm
Disp "The average is:",avg
EndPrgm

" ***** MAIN ROUTINE *****
inp_num()
calcavg()
out_avg()
```

### 6.4.2 Using a Loop ... EndLoop Loop

Suppose you don't know the number of values in advance. A For ... EndFor loop no longer works since you need to know how many times the loop will run. To solve this problem, select an impossible value, one that is extremely unlikely to appear in any set of values you enter. This signals the program that you want to end the loop. You then can use a Loop ... EndLoop structure.

Make a copy of the fore\_ave program and name it loop\_ave.

You can delete the input number subroutine, because we do not need to know the number of values to average in advance.

Modify the calcavg subroutine as shown to the right.

Rewrite the output subroutine to also display the number of values averaged.

Consider using concatenation to display the output on one line. Run the program.

```
loop_ave()
Prgm
" Calculate Avg with Loop EndLoop loop

" Calculate Average
Local calcavg
Define calcavg()=Prgm
" initialize variables
O»counter
O»total
Disp "Enter the values one at a time"
Disp "Enter -9999 to stop"
loop
Input "Enter next value:",nextx
if nextx ≠ -9999 then
  nextx+total»total
  1+counter»counter
else
  exit
EndIf
total/counter»avg
endloop
EndPrgm

" Output Average
Local out_ave
Define out_ave()=Prgm
Disp "The average of the "
&string(counter)&" values is: "
&string(avg)
EndPrgm

" ***** MAIN ROUTINE *****
calcavg()
out_ave()
```

```
whil_ave()
Prgm
" Calculate Avg with Loop EndLoop loop

" Calculate Average
Local calcavg
Define calcavg()=Prgm
" initialize variables
O»counter
O»total
Disp "Enter the values one at a time"
Disp "Enter ^9999 to stop"
Input "Enter the first value",nextx
While nextx≠^9999
  nextx+total»total
  1+counter»counter
  Input "Enter next value:",nextx
EndWhile
total/counter»avg
EndPrgm

" Output Average
Local out_ave
Define out_ave()=Prgm
Disp "The average of the "
&string(counter)&" values is: "
&string(avg)
EndPrgm

" ***** MAIN ROUTINE *****
calcavg()
out_ave()
```

### 6.4.3 Using a While ... EndWhile Loop

The loop to calculate the average in calcavg above also could be performed by a **While...EndWhile** loop.

Make a copy of the loop\_ave program and name it whil\_avg.

Modify the calcavg subroutine as shown to the right.

Note: The **While...EndWhile** structure differs from the Loop...EndLoop structure in that the value used to exit the **While...EndWhile** loop must be known before you enter the loop.

## 6.5 Nested Loops

### Running a Program More than Once

Let's look at a more complicated looping example.

Look at the program table\_7b (section 6.1 on page 55) the multiplication table of 7

You want to rewrite the program in such a way that it runs not just once for the number 7, but many times for any number the user can choose until the user decide to stop it.

To do this, embed the main body of the program inside a While ... EndWhile loop. Simply replace the main routine with statements to accomplish this task.

Make a copy of the existing program and name it whiltb\_n.

Modify the sub routine and main routine as shown to the right. Run the program.

This program runs as long as you enter a number  $\neq 0$  at the input prompt. As soon as you enter 0, the program ends.

You can do this with any program you've already written.

For example you might want the ABCForm() program in section 5.4.1 on page 47 to run for different values of a, b and c until you decide to stop it. To do this, embed the main body of the program inside a **While...EndWhile** loop. Simply replace the main routine with statements to accomplish this task.

Make a copy of the existing program abcform and name it abcform2. Modify the main routine as shown to the right and run the program.

This program runs as long as you enter y  at the **InputStr** prompt: **Again ? (y/n)**.

As soon as you enter n , the program ends. You can do this with any program you've already written.

```
whiltb_n()
Prgm
" The multiplication table of n:
" The new table_n
Local table(n)
Define table(n)=Prgm
  For k,1,9
    Disp string(k)&" x "&string(n)
    &" = "&string(k*n)
  EndFor
EndPrgm

" ***** MAIN ROUTINE *****
1→n
While n ≠ 0
  ClrIO
  table(n)
  Disp "To run once more, enter a number"
  Input "≠ 0, to stop enter 0.",n
EndWhile
EndPrgm

abcform2()
Prgm
" Solutions of Quadratic Equation
-----..
-----..
" ***** MAIN ROUTINE *****
ClrHome
"y"»again
While again="y"
  ClrIO
  enterdat()
  calcsolu()
  dispsolu()
  Pause
  ClrIO
  Loop
  InputStr "Again ? (y/n)",again
  If again="y" or again="n":Exit
EndLoop
EndWhile
EndPrgm
```

### Using the counter in nested loops.

Nested loops consist of an outer loop with one or more inner loops. Each time the outer loop is repeated, the inner loops are reentered, their loop-control expressions are reevaluated and all required iterations are performed.

The program `nestloop` and `sample` run as shown to the right demonstrate two nested for loops. The outer loop is repeated five times, for `i` equal 1, 2, 3, 4 and 5.

Each time the outer loop is repeated, the statement:

Disp " i j Enter"

displays the header string 'i j Enter',

the statement `Pause` lets the program wait until the user press `ENTER`

and the statement

```
Disp "outer "&string(i)
```

displays the string 'outer' and the value of i (the outer loop-control variable). Next the inner loop is entered, and its loop-control variable, j, is reset

to 1. The number of times the inner loop is repeated depends on the current value of  $i$ .

time the inner loop is repeated, the statement

```
Disp "inner "&string(i)&" "&string(j)
```

displays the string 'inner ' and the value of i and j.

The outer loop-control variable,  $i$ , is also the

expression whose value determines the number of repetitions of the inner loop. Although this is

perfectly valid, you cannot use the same variable as the loop-control variable of both an outer and an inner for loop in the same nest.

nestloop()

Prgm

" illustration of nested for loops

Clr|O

**For** i,1,5 " outer

Disp " i j Enter" " Heading

Pause

```
Disp "outer "&string(i)
```

**For** j,1,i " inner

```
Disp "inner "&string(i)&" "&string(j)
```

EndFor

**EndFor**

EndPrm

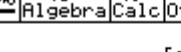
MODE	F1	F2	F3	F4	F5	F6
	Algebra	Calc	Other	Prgm	Clear	Ans
	i j	Enter				
outer	1					
inner	1 1					
	i j	Enter				
outer	2					
inner	2 1					
inner	2 2					
	i j	Enter				
outer	3					
inner	3 1					
inner	3 2					
inner	3 3					
	i j	Enter				
outer	4					
inner	4 1					
inner	4 2					
inner	4 3					
inner	4 4					
	i j	Enter				
outer	5					
inner	5 1					
inner	5 2					
inner	5 3					
inner	5 4					
inner	5 5					

### 6.5.1 Create and Fill a Matrix: An example

A matrix be made up of rows and columns filled with numbers. A matrix **A** with **r** rows and **c** columns turn into:

$$\begin{bmatrix} \mathbf{a}_{11} & \mathbf{a}_{12} & \mathbf{a}_{13} & \mathbf{a}_{14} & \dots & \mathbf{a}_{1c} \\ \mathbf{a}_{21} & \mathbf{a}_{22} & \mathbf{a}_{23} & \mathbf{a}_{24} & \dots & \mathbf{a}_{2c} \\ \mathbf{a}_{31} & \mathbf{a}_{22} & \mathbf{a}_{22} & \mathbf{a}_{22} & \dots & \mathbf{a}_{4c} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \mathbf{a}_{r1} & \mathbf{a}_{r2} & \mathbf{a}_{r3} & \mathbf{a}_{r4} & \dots & \mathbf{a}_{rc} \end{bmatrix} :$$

For example a 6 x 5 matrix on the TI-92:



The calculator screen displays a 6x5 matrix named `mat_x`. The matrix is filled with the values 1 through 30, arranged row by row. The top status bar shows function keys F1 through F6. The bottom status bar indicates '6 rows and 5 columns' and 'FUNC 1/30'.

	a11	a12	a13	a14	a15
a21	2	3	4	5	6
a31	7	8	9	10	11
a41	12	13	14	15	16
a51	17	18	19	20	21
a61	22	23	24	25	26

Its row and column index numbers specifies each element of a matrix. Therefore, the element  $a_{23}$  in row 2 and column 3 of the matrix **mat\_x** would be named **mat\_x[2,3]**

In working with any matrix, nested loops are very useful because you can use one loop to deal with the rows and another to deal with the columns.

Suppose you want to write a program to create a matrix. First, have the user input the number of rows and the number of columns for the matrix. Then, use nested loops to create the elements in the matrix. Begin by writing a program design such as one shown to the right.

### Solving the Problem

As you think through the solution, follow the steps below.

1. Create a matrix called **mat\_m** where each element in the matrix is the sum of the row number and the column number.

Let **nrows** be the number of rows in your matrix and **ncols** the number of columns. The user will input the values for both of these. Set **mat\_m[1, 1]** (the element in the first row and first column of the matrix **mat\_m**) to 1+1, **mat\_m[1,2]** to 1+2, **mat\_m[1,3]** to 1+3, and so on until you set **mat\_m[1,ncols]** to 1 +ncols. (See example to right.)

If you think of the second number in each bracket as a variable that assumes the successive values 1, 2, 3, ... up to **ncols**, you could combine all these statements into a single For ... EndFor loop.

2. Write the loop to fill the first row of the matrix as shown to the right. The number 1 in the second line of the loop is a fixed number-representing row 1 of the matrix.

**Note:** On the TI-92, you must use square brackets [ ] for matrices.

```
makeamat()
Prgm
© Create a Matrix
© with Appropriate Entries
Input Dimensions Subroutine
Create Matrix Subroutine
Display Matrix Subroutine
© MAIN ROUTINE
Invoke Subroutines
EndPrgm
```

$$\begin{bmatrix} 1+1 & 1+2 & 1+3 \\ 2+1 & 2+2 & 2+3 \\ 3+1 & 3+2 & 3+3 \\ 4+1 & 4+2 & 4+3 \end{bmatrix}$$

Above is **mat\_m** where **nrows=4** and **ncols=3**.  
The final **mat\_m** solution is:

$$\begin{bmatrix} 2 & 3 & 4 \\ 3 & 4 & 5 \\ 4 & 5 & 6 \\ 5 & 6 & 7 \end{bmatrix}$$

```
For j,1,ncols    ~ variable column no j
  1+j>mat_m[1,j] ~ fixed row number 1
EndFor
```

3. Now that row 1 is filled, you can write another loop for row 2. This loop is identical to the previous one except the fixed number 1 becomes a 2. This repeats until you reach the number of rows (nrows) needed. In the last loop, the fixed number could be replaced by a variable, nrows, where the user assigns the value to nrows. This is demonstrated in the getdims subroutine to the right, which also includes an Input statement to let the user assign a value to variable ncols.
4. Generalize the For ... EndFor loop to fill the i-th row by replacing the specific row number with the variable i.

This loop sets each element of the matrix mat\_m to be the current row number i plus the current column number j. Once this loop is finished, row i is complete.

5. Now nest this loop inside a loop that makes i take on the values from 1 to nrows.  
**Note:** *As previously discussed, the two nested loops must have different counter variables. The inner loop goes through a full set of repetitions for each repetition of the outer loop.*
6. To complete the subroutine, create the actual matrix using the newMat(nrows,ncols) command. This command creates a matrix with dimension nrows by ncols filled with zeroes in the calculator's memory.
7. Combine this subroutine with input and output subroutines to complete the program.

Now you are ready to enter the program and try it.

Start a new program in the Program Editor and name it makeamat.

Enter the program as shown to the right.

Note: It is important to understand in what order the matrix is actually getting its elements.  
Run the program.

To make sure you understand nested loops, reverse the two loops to see what happens to the program.

```
For j,1,ncols
  2+j>mat_m[2,j] " fixed row number 2
EndFor

" Get Dimensions of Matrix
Local getdims
Define getdims()=Prgm
Input "Enter number of rows:",nrows
Input "Enter number of columns:",ncols EndPrgm
```

```
For j,1,ncols " variable column no j
  i+j>mat_m[i,j] " variable row number i
EndFor
```

```
For i,1,nrows " variable row number i
  For j,1,ncols " variable column no j
    i+j>mat_m[i,j]
  EndFor
EndFor
```

```
makeamat()
Prgm
" Fill a Matrix with Specified Entries

" Get Dimensions of Matrix
Local getdims
Define getdims()=Prgm
Input "Enter number of rows:",nrows
Input "Enter number of columns:",ncols
EndPrgm
```

```
" Calculate Individual Matrix Entries
Local fillmat
Define fillmat()=Prgm
newMat(nrows,ncols)>mat_m
For i,1,nrows
  For j,1,ncols
    i+j>mat_m[i,j]
  EndFor
EndFor
EndPrgm
```

```
" Display Resulting Matrix
Local outdata
Define outdata()=Prgm
Disp "Your matrix is:",mat_m
EndPrgm
```

```
" ***** MAIN ROUTINE *****
getdims()
fillmat()
outdata()
EndPrgm
```

### 6.5.2 Print Out Zeros of an Expression: An example

Any command you can enter on the Home screen you can use in a program. Let's use the **zeros()** command to write a program that prints out all of the zeros of an expression.

The zeros() command creates a list of all zeros of the given expression. You can access the list elements by specifying them in brackets. Therefore, if zlist is the name of the list, zlist[1] is the first element, zlist[2] is the second element, etc. This is similar to specifying the elements of a matrix.

Think about the tasks to solve the problem and then write your program design.

Unlike earlier examples, there is no input subroutine because the expression for which you are solving is part of the zeros() command. The listzero subroutine assigns the list produced by the zeros() command to another list called zlist. Although the listzero subroutine is simple, the outlist subroutine requires some thought.

Start a new program and name it dispzero. Enter the listzero subroutine at the beginning of the program as shown to the right. Notice that you are finding the zeros of the expression  $x^6 - 14x^4 + 49x^2 - 36$ . To display the zeros, you must find the "length" of the list (*the number of zeros*) produced by the zeros() command so you will know how many values to display. The dim() command applied to a list returns the number of elements in the list. Use dim() when you write the output subroutine.

Enter the outlist subroutine.

Complete the program to display the zeros of  $x^6 - 14x^4 + 49x^2 - 36$  by adding the main routine. Run the program.

To find the zeros of another expression, change the expression in the zeros() command. In a later chapter, you will learn how to change the expression without having to rewrite the program.

```
dispzero()
Prgm
© Print Out Zeros of an Expression
Find Zeros Subroutine
Output Subroutine
© MAIN ROUTINE
Invoke Subroutines
EndPrgm
```

```
dispzero()
Prgm
" Prints Zeros of an Expression

" Find Zeros
Local listzero
Define listzero()=Prgm
zeros(x^6-14*x^4+49*x^2-36,x)»zlist
EndPrgm

" Print Out Zeros
Local outlist
Define outlist()=Prgm
dim(zlist) »n
For i,1,n
Disp zlist[i]
EndFor
EndPrgm

" ***** MAIN ROUTINE *****
listzero()
outlist()
```



## 6.6 Summary of Commands

**dim**(*list*)  $\Rightarrow$  *integer*  
Returns the dimension of *list*.  
For example: dim(0,1,2) [ENTER] returns: 3

**For** *counter, begin\_val, end\_val [,step\_size ]*  
*block of statements*

**EndFor** Executes the statements in *block of statements* iteratively for each value of **counter** from **begin\_val** upto (or downto) **end\_val**, in increments (or decrements) of **step\_size**.  
*counter* must not be a system variable.  
*step\_size* can be positive or negative, the default value is 1.

**InputStr** [*promptString*,] *var* pauses the program, displays *promptString* on the Program I/O screen, waits for you to enter an expression, and stores the expression in variable *var*.  
If you omit *promptString* , "?" is displayed as a prompt.

**Note:** The difference between **Input** and **InputStr** is that **InputStr** always stores the result as a string in such a way that " " are not required.

**int**(*expression*)  $\Rightarrow$  *integer*  
Returns the greatest integer that is less than or equal to the *expression*.  
For example: int(-2.7)[ENTER] returns: -3

**Loop**  
*block of statements*

**EndLoop** Repeatedly executes the statements in *block of statements* until an exit condition is satisfied.  
**Note** that the loop will be executed endlessly, unless a Goto or Exit instruction is executed within *block of statements*.

**mod**(*expression1,expression2*)  $\Rightarrow$  *expression*  
Returns the first argument modulo the second argument as defined by the identities:  
 $\text{mod}(x,0) \equiv x$   
 $\text{mod}(x,y) \equiv x - y \cdot \text{int}(x/y)$   
For example: mod(14,3)[ENTER] returns: 2  
 $\text{mod}(14,3) \equiv 14 - 3 \cdot \text{int}(14/3) = 14 - 3 \cdot 4 = 2$   
**Note:** See also remain() on page 133.

**newMat**(numRows,numCols)  $\Rightarrow$  *matrix*

Returns a matrix of zeros with the dimension numRows by numCols.

For example: newMat(7,2)**[ENTER]** returns: 
$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

**rand([n])**

$\Rightarrow$  *expression*

With no parameter *n*, returns the next random number between 0 and 1 in the sequence.

For example: rand()**[ENTER]** returns: 0.158

When an argument is positive, returns a random integer in the interval [1,n].

For example: rand(7)**[ENTER]** returns: 5

When an argument is negative, returns a random integer in the interval [-n, -1].

For example: rand(-7)**[ENTER]** returns: -3

**While condition**

*block of statements*

**EndWhile**

Executes the statements in *block of statements* as long as condition is true.

You must allow for the value *condition* to be changed within the *block of statements*.

**zeros(expression, var)**

$\Rightarrow$  *string*

Returns a list of candidate real values of *var* that make *expression* = 0.

For example: zeros(a\*x^2+b\*x+c,x)**[ENTER]** returns:

$$\left\{ \frac{-(\sqrt{-(4 \cdot a \cdot c - b^2)} + b)}{2 \cdot a}, \frac{(\sqrt{-(4 \cdot a \cdot c - b^2)} - b)}{2 \cdot a} \right\}$$

For some purposes, the result form for zeros() is more convenient than that of solve(). However, the result form of zeros() cannot express implicit solutions, solutions that require inequalities, or solutions that do not involve var.

## 6.7 Practical problems

Try the following exercises. Be sure to write a complete program using top-down program design, subroutines, and decision blocks.

### Problem 1

Replace in the nestloop program on page 61 the inner loop-control variable  $j$  into the outer loop variable  $i$ . Run the program and explain what is happening.

### Problem 2

The randcirc program on the right draws 25 circles with random centers and radii. It selects the center and radius of a circle randomly, with the TI-92 random number generator, and then uses the For...EndFor loop statement to draw the 25 random circles.

Modify the randcirc program to draw 10 random lines at a time. Set the values of the Window variables to  $xmin = -20$ ,  $xmax = 20$ ,  $ymin = -10$ , and  $ymax = 10$ . Generate random values of  $x0$  between  $-20$  and  $20$ , random values of  $y0$  between  $-10$  and  $10$ , and random values of  $m$  between  $-20$  and  $20$ . Finally, use the DrawSlp command to draw the line that passes through the point  $(x0, y0)$  and has slope  $m$ .

### Problem 3

Write a program that imitates the  $\max()$  command on the TI-92. Your program should read in a set of numbers and print out the largest value entered. Write the program two ways:

- 3a** The user enters the number of values desired and then enters the values.
- 3b** The user enters values until an "impossible" value is entered.

### Problem 4

An Identity matrix is a square matrix (dimension  $n$  by  $n$ ) with zeros everywhere except the main-diagonal elements are all equal to 1. An 4 by 4 identity matrix is shown on the right.

Write a program to create a  $k$  by  $k$  identity matrix. That is, the program asks the user for the number of rows in the matrix and then creates an identity matrix with that many rows and columns.

```
randcirc()
Prgm
  " Draw 25 Circles
  " with Random Centers and Radii

  " Set Window Variables subroutine
  Local setwin
  Define setwin()=Prgm
    0»xmin
    30»xmax
    0»ymin
    15»ymax
  EndPrgm

  " Draw circles subroutine
  Local drawcirc
  Define drawcirc()=Prgm
    " Creates a Loop
    For countc,1,25
      " Store a Number in xc,yc and rc
      rand(30)»xc
      rand(15)»yc
      rand(4)»rc
      " Draws the Circle
      Circle xc,yc,rc
    EndFor
  EndPrgm

  " Main Program
  clrdraw
  FnOff " Turns Off Selected Functions
  setwin()
  drawcirc()
```

F1	F2	F3	F4	F5	F6
Algebra	Calc	Other	PrgmIO	Clear	a-z...
matrix_i					
<div> <div>1</div> <div>0</div> <div>0</div> <div>0</div> </div> <div> <div>0</div> <div>1</div> <div>0</div> <div>0</div> </div> <div> <div>0</div> <div>0</div> <div>1</div> <div>0</div> </div> <div> <div>0</div> <div>0</div> <div>0</div> <div>1</div> </div>					
matrix_i					
CHAPTER6 RAD AUTO FUNC 1/30					

**Problem 5**

Modify the dispzero program (section 6.5.2 on page 64 ) in such a way that it prints out the list of zeros in reverse order (that is, starting with the nth zero and printing the first zero last).

**Problem 6**

Write a program that will find the smallest, largest, and average values in a collection of N numbers. Read in the value of N before reading each value in the collection of N numbers.

**Problem 7**

Write a program to generate a month of a yearly calendar. The program should accept the month, year and the day of the week for January 1 of that year (1 = Sunday, 7 = Saturday). The calendar should be printed in the following form:

```

January
      1
  2 3 4 5 6 7 8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31

```

**Problem 8**

Write a program to read in a collection of exam scores ranging in value from 1 to 100. Your program should count and print the number of outstanding scores (90 -100), the number of satisfactory scores (55 - 89), and the number of unsatisfactory scores (1-54). It should also display the category of each score. Test your program on the following data:

63 75 72 72 78 67 80 63 75 90 89 43 59 99 82 12 100

Modify your program to display the average exam score (a real number) at the end of the run.

**Problem 9**

The factorial of a positive integer quantity, n, is defined as  $N! = 1 \times 2 \times 3 \times 4 \times \dots \times (N-1) \times N$ . Thus  $2! = 1 \times 2$ ;  $3! = 1 \times 2 \times 3$ ;  $4! = 1 \times 2 \times 3 \times 4$ ; and so on.

Write a program to read in a positive integer and calculates a list of factorials.

**Problem 10**

The Fibonacci numbers form an interesting sequence in which each number is equal to the sum of the previous two numbers. In other words,

$$F_i = F_{i-1} + F_{i-2}$$

where  $F_i$  refers to the i-th Fibonacci number. The first two Fibonacci numbers are, by definition, equal to 1; i.e.,

$$F_1 = F_2 = 1.$$

Hence,

$$F_3 = F_2 + F_1 = 1 + 1 = 2$$

$$F_4 = F_3 + F_2 = 2 + 1 = 3$$

$$F_5 = F_4 + F_3 = 3 + 2 = 5$$

and so on.

Write a program to read in a positive integer N and determines the first N Fibonacci numbers. Test the program with N=23.

**Problem 11**

Generate the following "triangle of pascal" of digits, using nested loops. (Do not simply write out 10 multidigit strings).

```

      1
    1 2 1
  1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1

```

**Problem 12**

Generate the following "pyramid" of digits, using nested loops. (*Do not simply write out 9 multidigit strings*).

```

1
232
34543
4567654
567898765
67890109876
7890123210987
890123454321098
90123456765432109

```

**Problem 13**

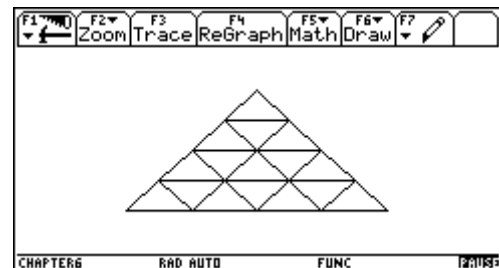
Develop a program that calculates the maximum amount of beer, which can be bought, with a given amount of money in a supermarket. The (empty) bottles can be returned to the supermarket. Starting point: a total amount in advance in your wallet, fixed costs for a bottle of beer and a fixed deposit for a bottle of beer.

**Problem 14**

Generate for example the following "pyramid of triangles".

Develop a program suchlike that:

- the "pyramid of triangles" is always completely visible on the screen,
- the program should accept the size (height and width) from the small triangles,
- and the user can determine in advance the amount of triangles on the baseline of the pyramid. (In this example thus 4 triangles).





## 7. Functions, Subroutines, Programs and Parameters

The introduction to subroutines (section 4.2 "Top-Down Design" on page 28) illustrated how to write separate program components –the subroutines- of a program. These subroutines, correspond to partial steps in a problem solution. In this chapter, you will learn a new type of program component namely functions. You will learn how to control the behavior of these functions, as well as the behavior of instructions, subroutines and programs, through the use of parameters.

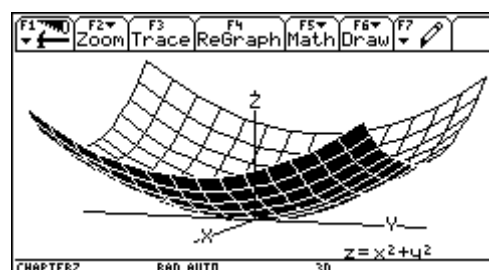
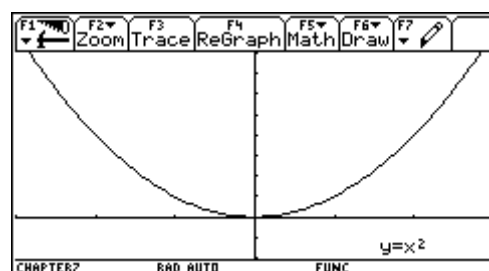
### 7.1 Functions

Functions are independent modules like subroutines, except that subroutines can return any number of results whereas a function always returns a single unique result.

For example the equation  $y = x^2$  defines  $y$  as function of  $x$  because for each  $x$  the square of  $x$  is unique. The equation  $y = x^2$  can be represented by the function  $f(x) = x^2$ . This means that applying the function  $f$  to the quantity  $x$  results in  $x^2$ . Thus  $f(2) = 4$ ,  $f(4) = 16$  and  $f(-3) = 9$ . This also applies to symbolic expressions, so  $f(a+3) = (a+3)^2 = a^2 + 6a + 9$ .

You can extend this notation to handle functions with multiple input.

For example the equation  $z = x^2 + y^2$  can be represented by the function  $g(x, y) = x^2 + y^2$ , in such a way that  $g(2, 3) = 2^2 + 3^2 = 13$ .



### Built-in functions

The TI-92 has a large number of built-in functions, that is, named functions, which have already been defined within the calculator. You can find a list in the Quick-Find Locator section of Appendix A in the TI-92 Guidebook. Some of

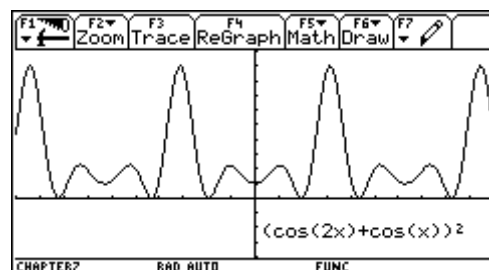
the more familiar examples are the trigonometric functions  $\sin()$ ,  $\cos()$ ,  $\tan()$ , the logarithmic functions  $e^x$ ,  $\ln()$ , and  $\log()$ , and single operators like  $x^{-1}$ ,  $\text{abs}()$ ,  $\sqrt{}$  and  $!$ .

Functions do not have to be mathematical, however. For example,  $\text{dim}()$  is a function since each list or string (the input) has a unique number of elements or length (the result).

### User-Defined Functions

Often, you need a function that is not built into the TI-92. If you want to graph an equation, for example, it is unlikely that the equation you want to graph is one of the built-in functions.

Suppose that you wish to define a new function  $f(x) = (\cos(2x) - 2\cos(x))^2$ ,  $x$  is called the independent variable or function parameter. In addition to defining functions in the Y= Editor, you also can define functions using the Store (») command or the Define command.



### The Store Command

To define a function using the Store command ( $\text{STO} \rightarrow$  key), you simply store the definition of the function into a name you choose for the function (with the appropriate number of input variables). Examples are shown to the right.

```
(cos(2*xx)-2*cos(xx))^2»f(xx)
xx^2+yy^2»g(xx,yy)
sin(xx)+yy^3+zz»g3(xx,yy,zz)
```

### The Define Command

With the Define command, the method is almost the same as with the Store command, except that the function and its definition are reversed and separated by an  $=$ . The examples above would be entered as shown to the right.

```
Define f(xx)=(cos(2*xx)-2*cos(xx))^2
Define g(xx,yy)=xx^2+yy^2
Define g3(xx,yy,zz)=sin(xx)+yy^3+zz
```

Although built-in functions and user-defined functions can cover most situations where you need a function, there are cases where a function is too complicated to define on a single line using Store (») command or the Define command.

Let's look at two such functions plus a piecewise function and see how you could create your own version of them.



### 7.1.1 Create an Absolute Value Function: An Example

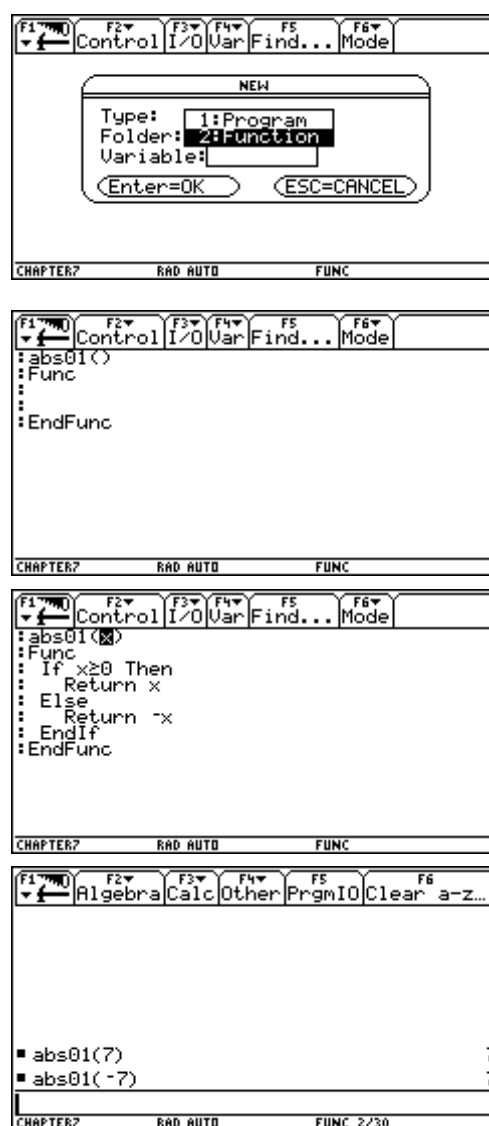
In section 5.2.1 (page 39), you wrote the absvalue program that imitated the absolute value function on the TI-92. Let's take those statements and actually create your own definition of absolute value.

There is no easy way to define the function as a single expression, since the absolute value of a number depends on whether the number is positive or negative. Therefore, you will create a new function.

1. Press **[APPS]** to start a new function variable in the Program Editor, select 7:Program Editor and then select 3:New.
2. Select 2:Function as the Type.
3. Type abs01 as the name of the new function variable.
4. Press **[ENTER]** **[ENTER]** to store the function name and display the new function template. The function name, **Func**, and EndFunc are shown automatically. Each line of the function begins with a colon (:).
5. Type the function lines as shown to the right. Be sure to include the **x** between the parentheses in the name of the function. The Return statement tells the calculator what the result of your function is and ends the function.
6. Return to the Home screen and run the abs01() function.

Be sure to include a value between the parentheses. For example, entering abs01(-7) returns 7.

**Note:** If the Return statement is skipped, the function automatically returns the last expression before EndFunc.



For example we copy the abs01() to abs02() and we modify the abs02() function as shown to the right.

In this case, if x is positive, the function returns x; but if x is negative, the If...Then statement is skipped and the function automatically returns the last expression before EndFunc.

```

F1 F2 F3 F4 F5 F6
Control I/O Var Find... Mode
:abs02(x)
:Func
: If x≥0 Then
:   Return x
: EndIf
: -x
: EndFunc
CHAPTER? RAD AUTO FUNC

```

### 7.1.2 Create an Factorial Function: An Example

Let's look at another, more complicated example of imitating a built-in function. The factorial function, represented by!, takes any positive integer (N) as input.

$$0! = 1$$

$$1! = 1$$

$$2! = 2 * 1$$

$$3! = 3 * 2 * 1$$

$$4! = 4 * 3! = 4 * 3 * 2! = 4 * 3 * 2 * 1$$

$$N! = N * (N - 1)! = N * (N - 1) * (N - 2)!$$

The result is the product of all positive integers less than or equal to N.

Some examples are given to the right.

A loop running from n downto 1 is required.

Start a new function variable in the Program Editor and name it fac01.

Enter the function lines shown to the right. Be sure to include the **n** in the name of the function.

Notice that the counter variable i and the return variable fc have meaning only inside the definition of fac01(n). They belonging just to the function and therefore need to be designated as local. If you don't designate them as local, you'll get an error message.

Return to the Home screen and run the fac01() function.

```

F1 F2 F3 F4 F5 F6
Control I/O Var Find... Mode
:fac01(n)
:Func
: Local fc,i
: i←fc @ initialize fc
: For i,n,1,-1
:   i*fc→fc
: EndFor
: fc @ Returns last expression
: EndFunc
CHAPTER? RAD AUTO FUNC

```

### Recursive functions (optional)

Instead of executing a For-loop to repeat the multiplication, we can use the statement:

$n * \text{fac}(n-1) \gg \text{fc}$

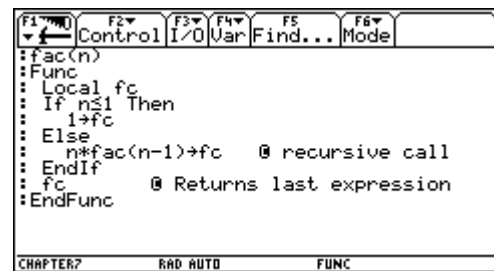
The statement contains a function designator,  $\text{fac}(n-1)$ , which calls function  $\text{fac}$  with an argument that is 1 less than the current argument. This function call is a *recursive call*.

If the argument in the initial call to  $\text{Fac}$  is 3, the following chain of recursive calls occurs:

$\text{Fac}(3) \rightarrow 3 * \text{Fac}(2) \rightarrow 3 * (2 * \text{Fac}(1))$

In the last of these calls,  $N$  is equal to 1, so the statement:  $1 \gg \text{fc}$  executes, stopping the chain of recursive calls.

When it finishes the last function call, the TI-92 must return a value from each recursive call, starting with the last one. The last call was  $\text{Fac}(1)$  and it returns a value of 1. To find the value returned by each call for  $N$  greater than 1, multiply  $N$  by the value returned from  $\text{Fac}(N-1)$ . Therefore the value returned from  $\text{Fac}(2)$  is  $2 * \text{the value returned from } \text{Fac}(1)$ ; the value returned from  $\text{Fac}(3)$  is  $3 * \text{the value returned from } \text{Fac}(2)$  etc.



## 7.2 Introduction to parameter lists

Parameter lists provide the communication links between the main program and its modules. Parameters make instructions, subroutines and functions more versatile because they enable a module to manipulate different data each time it is called.

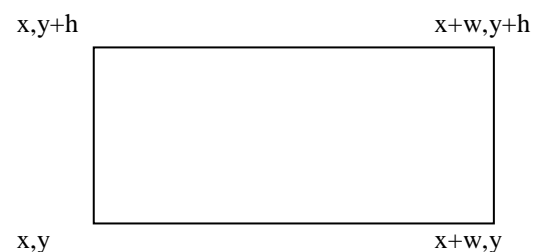
### Actual and Formal parameters.

Each subroutine with parameters has two parts: a **name** and an **actual parameter list**.

To illustrate the graphics subroutine `frame` on the right (or have a look at the house program on page 32) draws a rectangle from point  $(x1, y1)$ , with width  $(w1)$  and height  $(h1)$  on the screen. Its subroutine call statement: `frame(x,y,w,h)` consist of the subroutine name `frame`, and the actual parameter list  $(x,y,w,h)$ . The values of the four *actual parameters* are passed to subroutine `frame`, which draws the rectangle:

- a line from  $(x,y)$  to  $(x+w,y)$ ,
- a line from  $(x+w,y)$  to  $(x+w,y+h)$ ,
- a line from  $(x+w,y+h)$  to  $(x,y+h)$  and
- a line from  $(x,y+h)$  to  $(x,y)$ .

```
" Draws the Frame
Local frame
Define frame(x,y,w,h)=Prgm
  Line x,y,x+w,y
  Line x+w,y,x+w,y+h
  Line x+w,y+h,x,y+h
  Line x,y+h,x,y
EndPrgm
```



To draw another rectangle we need the subroutine call statement `frame(xx,yy,ww,hh)`. In each subroutine call, the programmer provides subroutine `frame` with four variables or values which represent the (x,y) left corner coordinates, the width and height of the rectangle on the screen.

Because the four coordinates can change each time `frame` subroutine is called, we must represent them somehow in the subroutine **Define** part.

To do this we use dummy names called *formal parameters*: `xs,ys,wi,he`

The formal parameter list shows the four formal parameters inside the subroutine to represent the x, y coordinates of the left corner (`xs,ys`) and the width (`wi`) and height (`he`) of the rectangle to be drawn.

### Correspondence Between Actual and Formal Parameters

Subroutine `frame` doesn't know what values it will receive until it is called.

The calling program in the **MAIN ROUTINE** passes the information needed by `frame` via the actual parameter list, matching each actual parameter with its corresponding formal parameter.

```

** Draws a Frame
Local frame
Define
frame(xs,ys,wi,he)=Prgm
  Line xs,ys,xs+wi,ys
  Line xs+wi,ys,xs+wi,ys+he
  Line xs+wi,ys+he,xs,ys+he
  Line xs,ys+he,xs,ys
EndPrgm

```

```

** ***** MAIN ROUTINE *****

```

```

frame(x,y,w,h)

```

```

frame(xx,yy,ww,hh)

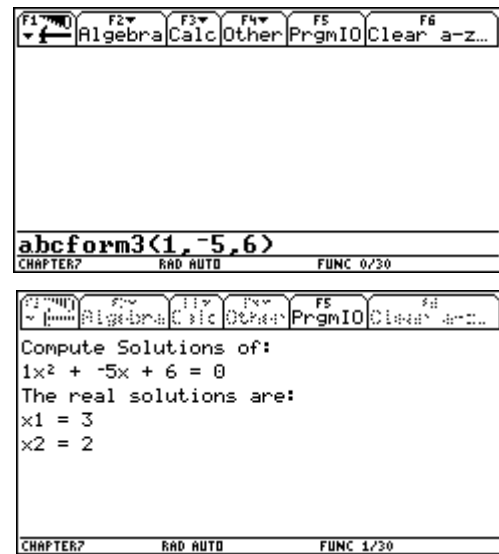
```

<u>actual</u>	corresponds to	<u>Formal</u>
x		xs
y		ys
w		wi
h		he
xx		xs
yy		ys
ww		wi
hh		he

Since subroutines and programs are similar, you also could use parameters for a program.

For example, instead of inputting the coefficients of a quadratic equation in the `abcform()` program (see section 5.4.1 on page 47), you could make them parameters. Running the new `abcform3(1,-5,6)` program would print out the solutions for the equation  $1x^2 - 5x + 6 = 0$ .

You can try this by eliminating the input subroutine `enterdat()`, changing the first line of the program to `abcform3(a,b,c)`, adding the parameters to the subroutine `calcsolu(a,b,c)` and `dispsolu(a,b,c)` and running the program.



```

abcform3(a,b,c)
Prgm
" Solutions of Quadratic Equation

" The CalcSolu subroutine
Local calcsolu
Define calcsolu(a,b,c)=Prgm
  -b/(2*a) → Part1
  b^2-(4*a*c) → Disc
  If Disc>0 Then
    √(Disc)/(2*a) → Part2
  ElseIf Disc<0 Then
    √(-Disc)/(2*a)*i → Part2
  EndIf
EndPrgm

" The Dispsolu subroutine
Local dispsolu
Define dispsolu(a,b,c)=Prgm
Disp string(a)&"x@ + "&string(b)&"x + "&string(c)&" = 0"
If Disc>0 Then
  Disp "The real solutions are:"
  Disp "x1 = "&string(Part1+Part2)
  Disp "x2 = "&string(Part1-Part2)
Elseif Disc=0 Then
  Disp "The only real solution is:"
  Disp "x = "&string(Part1)
Elseif Disc<0 Then
  Disp "The complex solutions are:"
  Disp "x1 = "&string(Part1+Part2)
  Disp "x2 = "&string(Part1-Part2)
EndIf
EndPrgm

***** MAIN ROUTINE *****
ClrIO
Disp Compute Solutions of:
calcsolu(a,b,c)
dispsolu(a,b,c)
EndPrgm

```

### 7.3 Summary of Commands

#### **DrawFunc** *expression*

Draws expression as a function of x, using x as the independent variable.

**Note:** Regraphing erases all drawn items.

#### **Func**

*block of statements*

#### **EndFunc**

Format required to define a multi(line)statement function.

#### **Local** *var1[,var2][,var3]*

Declares the specified vars as local variables. Those variables exist only during evaluation of a program or function and are deleted when the program or function finishes execution.

**Note:** Local variables save memory because they only exist temporarily. Also, they do not disturb any existing global variable values. Local variables must be used for temporarily saving values in a multiline function since modifications on global variables are not allowed in a function.

#### **Return**[*exp*]

Returns the expression *exp* as a result of the function. Use within a **Func...EndFunc** block.

For example: Define abs(x)=Func

: If  $x \geq 0$  Then

: Return x

: Else

: Return  $-x$

: EndIf

:EndFunc ENTER

abs(-7) ENTER "returns": 7

## 7.4 Practical problems

Try the following exercises. Be sure to write a complete function or program using top-down program design, functions and subroutines where needed.

### Problem 1

Define the following functions using multi-line functions.

- a) **Pos(x)** is a function that returns "Pos" if a number is positive and "Neg" if it is negative.

b) **Pieces(x)** has the value 
$$\begin{cases} x^2 & \text{if } x \leq 0 \\ 2x+1 & \text{if } 0 < x < 4 \\ 4-x & \text{if } x \geq 4. \end{cases}$$

- c) **Sumall(n)** returns the sum of all the positive integers squared from 1 to n:  $1^2+2^2+\dots+n^2$ .

- d) **Sumsome(a,b)** returns the sum of all the positive integers squared from a to b.

**Hint.** You can do this either by modifying your answer to (c) or by using the definition of sumall and noticing that sumsome(a,b) is sumall(b)-sumall(a-1).

- e) **Fibo(n)** returns the nth term of the Fibonacci sequence.

**Note:** The Fibonacci sequence is defined by  $F_1 = 1$ ,  $F_2 = 1$ , and  $F_3$  is the sum of the previous two terms, in such a way that the first few terms of this sequence are 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ....

- f) **Pow(x,y)** raises the real number x to the positive integer power y.

**Hint..** Write the definition using a loop, not the built-in exponentiation operation.

### Problem 2

Write a function  $C(N, R) = \frac{N!}{R!(N-R)!}$  that

returns the number of different ways  $R$  items can be selected from a group of  $N$  items.

### Problem 3

Add three subroutine calls to the main routine of the stick figure program in section **Fout!**

**Verwijzingsbron niet gevonden. Fout!**

**Verwijzingsbron niet gevonden.** on page 47.

These subroutines will draw a second and third

figure from the same size on the left and right side of the large figure.

#### Problem 4

Write a program using subroutines with parameters that will draw three houses of different sizes at different places on the screen. Your program should only have one set of housedrawing subroutines that you use three times with different values for the parameters. Ask the user for the size of each house (height and width) and the location of the bottom left corner of each house. (see section 4.2.1 Drawing a House: An example on page 28).

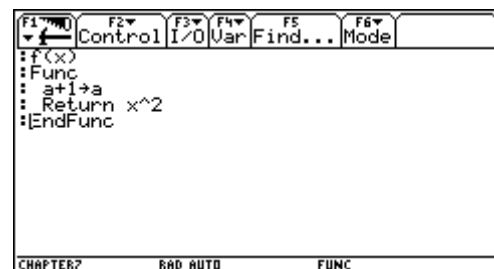
#### Problem 5

Rewrite the makeamat program (page 63) in such a way that it has two parameters corresponding to the number of rows and number of columns rather than asking the user to enter them.

#### Problem 6

To see why variables used within functions have to be local, use the function to the right. If  $a=2$  and  $b=3$ , first evaluate  $f(a)*f(b)$ , and then evaluate  $f(b)*f(a)$ . What happens in each case? Don't try it on your calculator, since it will give you an error.

**Note:** Assume that once  $x$  gets a value at the beginning of the function, it won't change during the execution of the statements in the function.



#### Problem 7

Write a recursive function that, given an input value of  $N$ , computes  $N + N-1 + \dots + 2 + 1$

#### Problem 8

Write a program jumpjack that let a stick figure jumps across the screen (see the stick figure program in section **Fout! Verwijzingsbron niet gevonden. Fout! Verwijzingsbron niet gevonden.** on page 47).



## 8. Data Types

With the "information age" comes the need to handle large quantities of data. Data is stored in a variety of ways, including lists, arrays, and tables. The way you store data usually depends on how you want to process it. In this chapter, you will learn about different kinds of data, methods of storing data, and programming techniques to process data in a variety of ways.

When you hear the word "data", you probably think of numerical data (statistics or numbers). In general "data" refers to any kind of input that a computer accept. This is also true for the TI-92. Data could include numbers, strings of characters, lists, matrices or other kinds of input.

The `getType()` command, when applied to a variable, tells you what type of data the variable represents. In this chapter, you will be working with six data types:

- Real numbers
- Expressions
- String Variables
- List Variables
- Data Variables
- Matrix Variables.

### 8.1 Real Numbers

This is the data type that you are most likely to think of when you see the word "data". On the TI-92, any variable, expression, or function that can be evaluated to a numerical result returns the type real number: "NUM" when you apply `getType()` to it.

1. From the Home screen, press `F6` `ENTER` to clear all values stored to one-character variables.
2. Enter the statements shown to the right. Each `getType()` returns the value "NUM".

### 8.2 Expressions

An expression is any statement that includes at least one variable that has not been assigned a

Statement	Result
<code>4+5→x</code>	9
<code>getType(x)</code>	"NUM"
<code>7→y</code>	7
<code>getType(y)</code>	"NUM"
<code>x+3*y→x</code>	30
<code>getType(x)</code>	"NUM"
<code>3.141592654→pi</code>	3.141592654
<code>getType(pi)</code>	"NUM"

value. Therefore, any variable quantity that cannot be evaluated as a single numerical value is an expression.

If the variables in the expression represent real numbers, you can use any of the same operations in the expression that you can use with real numbers. You also can use additional commands such as `expand()`, `solve()` and `factor()`.

1. On the Home screen, press `[F6][ENTER]` to clear all values stored to one-character variables.
2. Enter the statements shown to the right. Each `getType()` returns the value "EXPR".

### 8.3 String Variables

Any collection of characters surrounded by double quotation marks (") is considered a string even if the characters also make a sensible variable name or algebraic expression.

The operations you can use on strings are different from numerical operations.

1. On the Home screen, press `[F6][ENTER]` to clear all values stored to one-character variables.
2. Enter the statements shown to the right. To type `&`, press `[2nd]H`. Each `getType()` returns the value "STR".

Note: `&` represents concatenation. The strings separated by `&` are placed side by side in the list of statements in such a way that `t&t` has the value "tonton".

The functions `right(str,num)` and `left(str,num)` return the rightmost and leftmost num (number of) characters of str (string), respectively. Entering `mid(str,first,num)` returns the num characters of str starting in position first.

1. On the Home screen, enter the statements shown to the right.
2. Compare your results to those given.

Other useful string functions are:

Statement	Result
<code>x+3→y</code>	<code>x+3</code>
<code>getType(y)</code>	"EXPR"
<code>x+z→w</code>	<code>x+z</code>
<code>getType(w)</code>	"EXPR"
<code>abs(x)→y</code>	<code> x </code>
<code>getType(y)</code>	"EXPR"
<code>sin(x)→y</code>	<code>sin(x)</code>
<code>getType(y)</code>	"EXPR"

Statement	Result
<code>"m"→s</code>	"m"
<code>getType(s)</code>	"STR"
<code>"ton"→t</code>	"ton"
<code>getType(t)</code>	"STR"
<code>t&amp;t→u</code>	"tonton"
<code>getType(u)</code>	"STR"

Statement	Result
<code>"hello"→s1</code>	"hello"
<code>"there"→s2</code>	"there"
<code>s1&amp;" "&amp;s2→s3</code>	"hello there"
<code>right(s2,4)</code>	"here"
<code>left(s1,2)</code>	"he"
<code>mid(s2,2,3)</code>	"her"

- `expr()`  
Converts a string into a numerical or functional expression, then evaluates or executes it.
- `string()`  
Converts a numerical expression into a string.
- `dim()`  
Returns the length of a string.
- `inString(srcString,subString)`  
Returns the beginning position of substring if it occurs within `srcString`, and zero otherwise.

1. On the Home screen, enter the statements shown to the right.
2. Compare your results to those given.

Statement	Result
<code>s3</code>	"hello there"
<code>inString(s3,"the")</code>	7
<code>inString(s3,"he")</code>	1
<code>inString(s3,"them")</code>	0
<code>"3*2+5"→s4</code>	"3*2+5"
<code>expr(s4)</code>	11
<code>string(3+4/2)</code>	"5"
<code>string(b+5)</code>	"b+5"
<code>string(expr(s4))</code>	"11"
<code>expr("dim(sl)")</code>	5

### 8.3.1 Create a Count Letters Program: An Example

To demonstrate the use of some of the string functions, let's write a subroutine that counts the number of times the letters **th** appears together in a string. You can use `inString()` to find the first **th**, but then you must remove the first part of the string to avoid counting the same **th** over and over. With the `right()` function, you can keep only the part of the string you haven't examined yet. Think about the tasks to solve the problem and then write your program design.

```

Local countstr
Define countstr()=Prgm
  While String Not Empty
    Search for Next Substring
    Increase Count
    Save Rest of String
  EndPrgm

```

A subroutine that could result from the program design is shown to the right.

1. Start a new program in the Program Editor and name it countth.
2. Enter the countstr subroutine as shown to the right.

```
Count_th()
Prgm
" ***** SUBROUTINE *****
" Count "th" in a sentence
Local countstr
Define countstr()=Prgm
str»tempstr
0»count
dim(tempstr)»lenght
While lenght>0
  inString(tempstr,"th")»nextpos
  If nextpos>0 Then
    count+1»count
  Else
    Exit
  EndIf
  right(tempstr,lenght-nextpos-1)»tempstr
  dim(tempstr)»lenght
EndWhile
EndPrgm
```

3. Add input and output subroutines.

Tip: Instead of using the Input command to enter strings, which means you must include quotation marks on all input strings, you can use the InputStr command, which allows you to enter the input strings without the quotation marks.

```
Local indat
Define indat()=Prgm
ClrHome
ClrIO
InputStr "Please give a sentence",str
EndPrgm

Local out
Define out()=Prgm
Disp "", "The number of th ="&string(count)
EndPrgm
```

Note: If you enter the string "This is the end," the program counts only one th since Th with a capital T is considered to be a different string.

4. Try the program with a variety of input strings.

```
" ***** MAIN ROUTINE *****
indat()
countstr()
out()
DelVar str,count,lenght,tempstr,nextpos
EndPrgm
```

## 8.4 List Variables

Lists are collections of other kinds of data. They are represented as a series of data values enclosed in braces. For example, {a,8,"hi",\_, -1.88} is a list.

1. On the Home screen, press **[F6][ENTER]** to clear all values stored to one-character variables.
2. Enter the statements shown to the right. Each `getType()` returns the value "LIST".

Note: You must enter a list using commas, but its value is shown with spaces.

Statement	Result
{1,4,9,16}»L1	{1 4 9 16}
getType(L1)	"LIST"
{1,4,8,16}»L2	{1 4 8 16}
getType(L2)	"LIST"
{1,4,6,8,10}»L3	{1 4 6 8 10}
getType(L3)	"LIST"

You can perform mathematical operations on lists.

1. On the Home screen, enter the statements shown to the right.
2. Compare your results with those given.

Note: The `dim()` command works on both lists and strings. This is also true of `right()`, `left()` and `mid()`

You can access individual members of a list by specifying a position within brackets as shown to the right.

1. On the Home screen, enter the statements shown to the right.
2. Compare your results with those given.

Statement	Result
L1	{1 4 9 16}
L2	{1 4 8 16}
L3	{1 4 6 8 10}
L1+L2	{2 8 17 32}
\$(L1)	{1 2 3 4}
L1/L2	{1 1 9/8 1}
L1-L3	Error: Dimension mismatch
L2^3	{1 64 512 4096}
L2^(-1)	{1 1/4 1/8 1/16}
dim(L1)	4
L1[4]	16
L2[3]	8
L2[5]	Error: Dimension

#### 8.4.1 Findkey, a List Variables Program: An Example

Now try writing a program that uses lists. The program in section 6.5.2 on page 64 generates a list of the zeros of an expression. This time write a program to search a list of values for a specified number (often called the "key").

The program design might be as shown to the right.

Your program should tell the user the first position that contained the number (key). If key isn't found, display Key not found or else return a position of 0.

```
findkey()
Prgm
• Get Input
• Generate List
• Search List for Specified Number
• Print Location
~ MAIN ROUTINE
Invoke Subroutines
EndPrgm
```

1. Start a new program in the Program Editor and name it findkey.
2. The subroutine to get the input asks the user to enter how many numbers are in the list, the largest integer allowed in the list, and the number for which to search.

Enter the indata subroutine as shown to the right.

3. Now let's generate a random list using rand(n) to produce random values between 1 and n. The newList() command is like the newMat() command you used in section 6.5.2 except it creates a list instead of a matrix. Enter the makelist subroutine after the indat subroutine.

4. Next, write a subroutine to compare the key to each list element until the key is found. One possible approach is to use a While ... EndWhile loop.

Enter the keyhunt subroutine. To type  $\neq$ , press  $\boxed{2\text{nd}}\boxed{V}$  or  $\boxed{\div}\boxed{=}$ .

Examine the keyhunt subroutine carefully. Do you see a bug in it. What happens if the key isn't in the list?

5. If you run this subroutine with a key that isn't in the list, you get an error. The loop runs until **Position** is equal to length. Since key isn't found, Position would become length+1. The next repetition of the loop tries to examine list1[length+1], which doesn't exist, so you get a dimension error. To avoid this difficulty, add two conditions to your loop: one to check whether or not the key is found and one to make sure you haven't gone beyond the end of the list. Modify the keyhunt subroutine as shown on the right. Now the loop ends if either the key is found or the end of the list is passed.

6. Finally, you need to tell the user if the key was found, and if so, where. You can tell that the key was found if position  $\leq$  length, since position equals one more than length if the key isn't in the list. Enter the output subroutine, outdat.

```
findkey()
```

```
Prgm
" Get input
Local indat
Define indat()=Prgm
Input "Numbers in your list: ",length
Input "Largest integer in list:",maxn
Input "Number you wish to find: ",key
EndPrgm
```

```
" Generate the list
Local makelist
Define makelist()=Prgm
newList(length)=list1
For i,1,length
  rand(maxn)»list1[i]
EndFor
EndPrgm
```

```
" look for key
Local keyhunt
Define keyhunt()=Prgm
1»position
While list1[position]≠key and position≤length
  position+1»position
If position>length
  Exit
EndWhile
EndPrgm
```

```
" print out results
Local outdat
Define outdat()=Prgm
If position≤length Then
  Disp "Key found at position:"&string(position)
Else
  Disp "Key not found in list"
EndIf
```

7. Add the main routine to call the above subroutines.
8. Run the findkey program with a variety of inputs, including once where the key is not in the list.

*Hint. You can make sure that the key is not in the list by making the list entries in the range 1 to maxn and setting the key to be maxn+1.*

This program searches for a number in a list of random numbers. The list can be letters as well as numbers, and you can search for a character or a string of characters.

```

***** MAIN ROUTINE *****
ClrHome
ClrIO
indat()
makelist()
keyhunt()
ClrIO
outdat()
Pause
DispHome

EndPrgm

```

## 8.5 Data Variables

The basic framework of findkey can be modified to solve other searching problems. The final example in section 8.7.1 on page 96 is just such a program.

A data variable is a list of lists, that is, a collection of lists, which can be different lengths. You can construct a data variable using the NewData command, which specifies the name of the new variable followed by the names of the lists that make it up. You cannot display a data variable directly on the Home screen, but you can display the individual lists that make it up.

1. On the Home screen, enter the statements shown to the right.  
The `gettype()` command returns the value DATA.
2. The statements shown to the right illustrate how to access the elements of a data variable. Compare your results to the ones shown. Since `Dat1[2]` and `Dat1[3]` are lists you can access specific elements of these lists by adding a second bracketed number.

Statement	Result
{1, 2, 3, 4}»L1	{1 2 3 4}
{3, 4, 5, 6}»L2	{3 4 5 6}
{"a","b","c"}»L3	{"a" "b" "c"}
NewData Dat1,L1,L2,L3	Done
getType(Dat1)	"DATA"

Statement	Result
Dat1	Error: Non-algebraic variable in expression
Dat1[1]	{1 2 3 4}
Dat1[3]	{"a" "b" "c"}
Dat1[2][3]	5
Dat1[3][2]	"b"

## 8.6 Matrix Variables

A matrix is a rectangular array of numbers (see section 6.5.1: "Create and Fill a Matrix: An example" on page 61). In some ways it resembles a data variable, except that all the lists that make it up must be the same length, and unlike a data variable, a matrix can be displayed and operated on.

The mathematical operations that are available for matrices are addition (+), subtraction (-), multiplication (\*), matrix multiplication (.\*), division (/) and exponentiation (^). In each operation beginning with a period (.) the corresponding elements of the matrices are combined using that operation.

1. On the Home screen, define three matrices as shown to the right.  
The elements in the first pair of brackets define the first row of the matrix, the second set of elements is the second row, etc.
2. You can perform mathematical operations on the matrices. On the Home screen, enter the statements to the right. Compare your results with those given.

In addition to these operations, you also can raise square matrices (but not nonsquare matrices) to positive and other negative powers. Positive powers correspond to repeated matrix multiplication, the -1 power equals the inverse (if it exists), and other negative powers correspond to repeated matrix multiplication of the inverse.

To multiply two matrices, the number of elements in each row of the first matrix must equal the number of elements in each column of the second matrix. Therefore,  $mt1 * mt2$  is undefined, since  $mt1$  has three columns but  $mt2$  only has two rows.

*Note: If you accidentally leave out the period, addition and subtraction still work the same way, but division and exponentiation are undefined. Matrix multiplication is represented by " \* ", while entry-by-entry multiplication is represented by " . \* ".*

```
[[1,2,3][4,5,6]]»mt1
[[1,3,5][2 4 61 ]»mt2
[[3,2,1][2,1,3][3,1,2]]»mt3
```

Statement	Result
mt1	$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$
mt2	$\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$
mt3	$\begin{bmatrix} 3 & 2 & 1 \\ 2 & 1 & 3 \\ 3 & 1 & 2 \end{bmatrix}$
mt1.+mt2	$\begin{bmatrix} 2 & 5 & 8 \\ 6 & 9 & 12 \end{bmatrix}$
mt1.-mt2	$\begin{bmatrix} 0 & -1 & -2 \\ 2 & 1 & 0 \end{bmatrix}$
mt1.*mt2	$\begin{bmatrix} 1 & 6 & 15 \\ 8 & 20 & 36 \end{bmatrix}$
mt1*mt3	$\begin{bmatrix} 16 & 7 & 13 \\ 40 & 19 & 31 \end{bmatrix}$
mt1./mt2	$\begin{bmatrix} 1 & 2/3 & 3/5 \\ 2 & 5/4 & 1 \end{bmatrix}$
mt1.^mt2	$\begin{bmatrix} 1 & 8 & 243 \end{bmatrix}$



*Hint: To multiply two matrices, you multiply each row of the first matrix by each column of the second matrix. Multiplying a row by a column means multiplying the corresponding elements and then summing all the products.*

### 8.6.1 Matrix multiplication: An Example

To multiply two matrices, you need an outer loop to take care of the rows, an inner loop to take care of the columns and an "inner" inner loop to handle the actual multiplication.

This innermost loop is shown to the right. The dimensions of mat1 are  $R1 \times C1$  and the dimensions of mat2 are  $R2 \times C2$ , where  $C1=R2$ . The multiplication result is stored into mat3.

```
For k,1,C1
mat3[i,j]+mat1[i,k]*mat2[k,j]»
mat3[i,j]
EndFor
```

This loop multiplies the corresponding entries in row i of mat1 and column j of mat2 and adds them into position [i,j] of mat3

```
newMat(R1,C2)»mat3
For i,1,R1
For j,1,C2
For k,1,C1
mat3[i,j]+mat1[i,k]*mat2[k,j] »mat3[i,j]

EndFor
EndFor
EndFor
```

Putting this loop inside two outer loops that control the rows of mat1 and the columns of mat2 gives you a subroutine that multiplies mat1 by mat2.

Try building these statements into a program and applying it to mt1 and mt3 above. It will be helpful to apply the statements yourself to see what is happening.

## 8.7 Data Types and Databases

Now, let's write a program that uses some of the data types and programming ideas discussed in the previous sections. The next program uses a database, a collection of values organized in a specific structure, to answer a variety of questions.

### Creating the Database

You can store information in a database with the Data/Matrix Editor.

Suppose you want to create a database of names, addresses, telephone numbers, and balances owed to your company. You can store the first name, last name, street address, city, country, zip-code and telephone number as separate elements of a

list with one list per person. The lists are stored as a data variable.

1. Press **[APPS]** to start a new data variable in the Data/Matrix editor, select 6:Data/Matrix Editor, and then select 3: New. Press **[↓]** **[↓]** and type datafile as the name of the new variable.
2. Press **[ENTER]** **[ENTER]** to store the variable name and display an empty variable in the Data/Matrix Editor.
3. The cursor is in the first entry of the first column (c1). You are now ready to enter data.
4. Type each of the elements as shown, including the quotation marks. Press **[ENTER]** after each one.

You should now have seven items of information in the first seven rows (cells) of the first column.

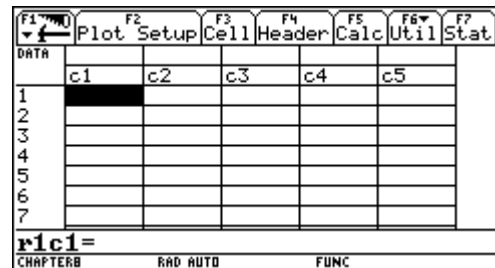
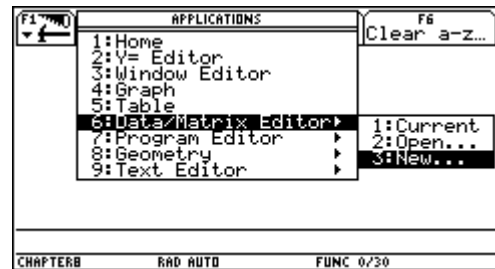
To edit any mistakes or change an entry, use the cursor key to go to the required cell.

*Note: If all of the information does not fit in a cell, the first five characters are displayed, followed by an ellipsis*

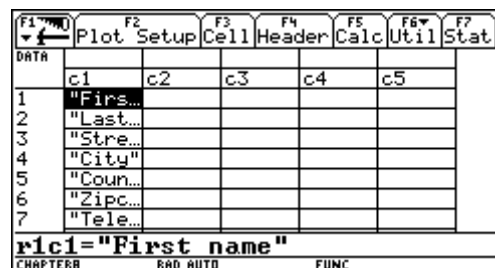
5. When you have finished typing in all the information, press **[2nd]****[QUIT]** to exit the Data/Matrix Editor and return to the Home screen.

"Ton"	"Lilian"
"Maree"	"van Erk-Frijters"
"Kruburg 74"	"De Koppele 395"
"Eindhoven"	"Eindhoven"
"Netherlands"	"Netherlands"
"5632 PJ"	"5632 LN"
" +31-40-2421694"	" +31-40-2426010"

"Martijn"	"end"
"van Dongen"	"end"
"Rachelsmolen 1"	"end"
"Eindhoven"	"end"
"Netherlands"	"end"
"5612 MA"	"end"
" +31-40-2605344"	"end"



"First name"  
 "Last name"  
 "Street"  
 "City"  
 "Country"  
 "Zip-code"  
 "Telephone"



*Note: An empty cell: is a pair of quotation marks with a space between them.*

### 8.7.1 Access information in a NAC database: An Example

#### Problem

Create a program, which can access the database created above, and retrieve the information when given a customers last name. Assume each last name is unique.

To solve this problem we will use the software development method described in chapter 4.

#### Analysis

The first step in the analysis of this problem is to divide the main problem into smaller subproblems.

A possible analysis could give these three subproblems:

- Input the customers last name:

This is a problem we have seen before and can be easily solved.

- Access the database to find the customer:

This problem is more difficult and therefore has to be studied more deeply.

First we have to find a method to access the database.

Second we must check if the customers last name coincides with the *field* "last name" in the database. To do this we need a search routine which checks all entries in the "last name" field and compare the result with the entered customer name.

Third we continue the search until the name is found in the database or the end of the database is reached. If the customer is found the location of the customer has to be saved into memory to display the information in a later stage.

- Retrieve and display the customer information.

The search can result into two solutions. Either the name is found and the customer information has to be displayed (location is saved) or the name has not been found (location is not saved)

and a message saying "name not found" has to be displayed.

Looking back at the subproblems already some variables can be defined for later use in the program:

Lastname	Customers last name
Found	Customer found in database
Chuzdata	Database
Loc	Location of customer information in database.

### Design

A possible design of the program is shown to the right.

### Implementation

Now we will write our program Chuzcus() in the program editor. First type the program listing for the input customer subroutine.

#### Input customer subroutine.

We will call the subroutine Inputnam(). When asking for the customer name we have to be sure that the variable "lastname" will be off the type string. Using the Inputstr command can do this.

#### Find customer in database

The subroutine for this section we will call findcust(). To search the database we must make use of a loop. Because we do not know the size of the database we cannot use the For..Endfor statements. Instead we will use the While..EndWhile statements.

Before we can enter the While loop, we first have to initialize the variables found and loc (location) to respectively false (i.e. customer name not found) and 1 (first column). The while loop will continue to until the customer is found or until the end of the database is reached (if statement combined with Exit).

```
Chuzcus()
Prgm
" This program displays information about a customer
by retrieving information from a database
```

```
***** Subroutines *****
Input customer name
Find customer in database
Display customer information
```

```
***** Main Routine *****
Invoke subroutines
```

```
EndPrgm
```

```
" Input customer name
Local inputnam
Define inputnam()=Prgm
Inputstr "Enter customers last name",lastname
EndPrgm
```

```
" Find customer in database
Local Findcust
Define Findcust()=Prgm
false»Found
0»loc
While not(found)
loc+1»loc
If lastname=(chuzdata[loc])[2] then
true»Found
Elseif (chuzdata[loc])[2]="end" then
Exit
Endif
EndWhile
EndPrgm
```

### Display Information

The subroutine for this section is called Dispinfo(). The two possible solutions to the search subroutine (found=true or found=false) are now evaluated with an If..Then statement. In case of found=false the message "customer not found" will be displayed. In case of found=true the customer information will be displayed on the screen using the variable location to extract the information for address etc. from the database chuzdata.

```

" Display information
Local Dispinfo
Define Dispinfo()=Prgm
If Found="no" then
  Disp "Customer not found"
Else
  Disp "First name:"&(chuzdata[loc])[1]
  Disp "Last name:"&(chuzdata[loc])[2]
  Disp "Address:"&(chuzdata[loc])[3]
  Disp "Zip code:"&(chuzdata[loc])[6]
  Disp "City:"&(chuzdata[loc])[4]
  Disp "Country:"&(chuzdata[loc])[5]
  Disp "Telephone:"&(chuzdata[loc])[7]
  Disp ""
Endif
EndPrgm

```

### Main Routine

The main routine is for the program Chuzcus() is shown to the right.

```

" ***** MAIN ROUTINE *****

Inputnam()
Findcust()
Dispinfo()

```

### Testing

The program can now be tested using several different customer names including names listed and names not listed in the database.

Possible errors are:

- typing errors,
- no database called chuzdata present,
- no "end" column to indicate the end of the database,
- too few Endprgm, Endwhile or Endif statements,
- Etc..

## Maintenance

The last step in solving the problem is maintaining the program. This step includes making the program look good, cleaning up variables and making the program more user friendly. You can also optionally test for the existence of the database.

1. Making the program looks good.  
This can be done by adding several ClrIO statements at tactical positions in the main routine.
2. Cleaning up variables.  
Add the Delvar-statement to the program
3. Making the program more user friendly.  
This part of the maintenance can go into several directions. For instance you can make look better, but this has been done already in the first option. You can also think about making it possible to search for another customer without leaving the program. A possible solution (as a subroutine) is given to the right. This subroutine is combined with the Loop..EndLoop statements in the main routine.
4. A test can be done using the Gettype(var) command to see if the variable Chuzdata is of the type "Data". If not the program will stop.

```

" Restart program subroutine
Local Again
Define Again()=Prgm
Loop
Inputstr "Again (y/n)",yesno
If yesno="y" or yesno="Y" then
Exit
elseif yesno="n" or yesno="N" then
Delvar yesno,loc,found,lastname
Stop " Stops program
Endif
Endloop
EndPrgm

```

" \*\*\*\*\* MAIN ROUTINE \*\*\*\*\*

```

Loop
ClrIO
Inputnam()
Findcust()
Clrlo
Dispinfo()
ClrIO
Again()

```

```

" Check to see if chuzdata exists
Local Exist
Define Exist()=Prgm
if gettype(chuzdata)≠"DATA" then
Disp "Database Chuzdata has not been
found or"
Disp "variable Chuzdata is not a database"
Disp " "
Stop
Endif
EndPrgm

```



## The final program

The final program listing is shown below.

```

chuzcus()
Prgm
" This program displays information about a customer
by retrieving information from a database

" ***** SUBROUTINES *****

" Input customer name
Local inputnam
Define inputnam()=Prgm
  Inputstr "Enter customers last name",lastname
EndPrgm

" Find customer in database
Local Findcust
Define Findcust()=Prgm
  false»Found
  0»loc " counter for location (column)
  While not(found)
    loc+1»loc
    If lastname=(chuzdata[loc])[2] then
      true»Found
    Elseif (chuzdata[loc])[2]="end" then
      Exit
    Endif
  EndWhile
EndPrgm

" Display information
Local Dispinfo
Define Dispinfo()=Prgm
  If Found=false then
    Disp "Customer not found"
    Disp "Press enter to continue"
    Pause
  Else
    Disp "First name: "&(chuzdata[loc])[1]
    Disp "Last name: "&(chuzdata[loc])[2]
    Disp "Address: "&(chuzdata[loc])[3]
    Disp "Zip code: "&(chuzdata[loc])[6]
    Disp "City: "&(chuzdata[loc])[4]
    Disp "Country: "&(chuzdata[loc])[5]
    Disp "Telephone: "&(chuzdata[loc])[7]
    Disp "Press Enter to continue"
    Pause
  Endif
EndPrgm

" Restart program subroutine
Local Again
Define Again()=Prgm
  Loop
    Inputstr "Again (y/n)",yesno
    If yesno="y" or yesno="Y" then
      Exit
    elseif yesno="n" or yesno="N" then
      Delvar yesno,loc,found,lastname
      Stop " Stops program"
    Endif
  Endloop
EndPrgm

" Check to see if chuzdata exists
Local Exist
Define Exist()=Prgm
  if gettype(chuzdata)≠"DATA" then
    Disp "Database Chuzdata has not been found or"
    Disp "variable Chuzdata is not a database"
    Disp " "
    Stop
  Endif
EndPrgm

" ***** MAIN ROUTINE *****

ClrIO
Exist()
Loop
  ClrIO
  Inputnam()
  Findcust()
  Clrlo
  Dispinfo()
  ClrIO
  Again()
EndLoop

EndPrgm

```



## 8.8 Summary of Commands

- dim(list)**  $\Rightarrow$  *integer*  
Returns the dimension of list.  
For example: dim(0,1,2) [ENTER] returns: 3
- dim(matrix)**  $\Rightarrow$  *list*  
Returns the dimensions of matrix as a two-element list {rows, columns}.  
For example: dim([1,-1,2;-2,3,5]) [ENTER] returns: {2,3}
- dim(string)**  $\Rightarrow$  *integer*  
Returns the number of characters contained in character string *string*.  
For example: dim("Hello") [ENTER] returns: 5  
dim(" Hello"&" there") [ENTER] returns: 11
- expr(string)**  $\Rightarrow$  *expression*  
Returns the character string contained in *string* as an expression and immediately executes it.  
For example: expr("3+4+x^5+2x+x") [ENTER] returns:  $x^5+3x+7$
- getType(var)**  $\Rightarrow$  *String*  
Returns a string indicating the TI-92 data type of variable *var*.  
For example: {1,2,3}  $\rightarrow$  temp [ENTER] returns: {1 2 3}  
getType(temp) [ENTER] returns: "LIST"

### Data Type Variable Contents

"DATA"	Data type
"EXPR"	Expression (includes complex/ arbitrary/ undefined, $\infty$ , $-\infty$ , TRUE, FALSE, pi, $e$ )
"FUNC"	Function
"LIST"	List
"MAT"	Matrix
"NONE"	Variable does not exist
"NUM"	Real number
"PIC"	Picture
"PRGM"	Program
"STR"	String
"TEXT"	Text type
"VAR"	Name of another variable

- inString(srcString,subString)**  $\Rightarrow$  *integer*  
Returns the character position in string *srcString* at which the first occurrence of string *subString* begins.

For example: `inString("Ton is there","the")``[ENTER]` returns: 8

**left**(*srcString*[,*num*])

⇒ *string*

Returns the leftmost *num* characters contained in character string *srcString*.

For example: `left("Martijn",2)``[ENTER]` returns: "Ma"

If you omit *num*, returns all of *srcString*.

**left**(*list1*[,*num*])

⇒ *list*

Returns the leftmost *num* elements contained in *list1*.

For example: `left({1,2,-3,4},3)``[ENTER]` returns: {1 2 -3}

If you omit *num*, returns all of *list1*.

**mid**(*srcString*,*start*[,*count*])

⇒ *string*

Returns *count* characters from character string *srcString*, beginning with character number *start*.

For example: `mid("Hello there",1,5)``[ENTER]` returns: "Hello"

If *count* is omitted or is greater than the dimension of *srcString*, returns all characters from *srcString*, beginning with character number *start*.

For example: `mid("Hello there",2)``[ENTER]` returns: "ello there"

*count* must be  $\geq 0$ . If *count* = 0, returns an empty string.

**mid**(*srcList*,*start*[,*count*])

⇒ *list*

Returns *count* elements from *srcList*, beginning with element number *start*.

For example: `mid({9,8,7,6},2,2)``[ENTER]` returns: {8 7}

If *count* is omitted or is greater than the dimension of *srcList*, returns all elements from *srcList*, beginning with element number *start*.

For example: `mid({9,8,7,6},3)``[ENTER]` returns: {7 6}

*count* must be  $\geq 0$ . If *count* = 0, returns an empty string.

**newList**(*numElements*)

⇒ *list*

Returns a list with a dimension of *numElements*. Each element is zero

For example: `newList(7)``[ENTER]` returns: {0 0 0 0 0 0 0}

**newMat**(*numRows*,*numCols*)

⇒ *matrix*

Returns a matrix of zeros with the dimension *numRows* by *numCols*.

For example: `newMat(7,2)``[ENTER]` returns: 
$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

**right**(*list1,num*) $\Rightarrow$  *list*Returns the rightmost *num* elements contained in *list1*For example: right({1,2,3,4,5,6,7},3) returns: {5 6 7}**right**(*srcString,num*) $\Rightarrow$  *string*Returns the rightmost *num* characters contained in character string *srcString*.For example: right("Hello Ton",2) returns: "on"**string**(*expression*) $\Rightarrow$  *string*Simplifies *expression* and returns the result as a character string.For example: string(0.1234567)  returns: "0.1234567"string(1+2) returns: "3"string(sin( $\pi/6$ ) +7)  returns: "15/2"string(7+45) returns "52". The string "52" represents the characters "5" and "2", not the number 52.

## 8.9 Practical problems

Try the following exercises. Be sure to write a complete function or program using top-down program design, functions and subroutines where needed.

### ***Problem 1***

Write a program named reverse that takes any string and writes it out backward.

### ***Problem 2a / 2b***

Write a program that takes any string and determines whether it is a palindrome.

Note: A *palindrome* is a word or string like "madam" or "redivider" that reads the same forwards and backwards.

***Problem 2a:*** use "reverse" from ***Problem 1***

***Problem 2b:*** use "compare characters"

### ***Problem 3***

Write a program to create a list arblst whose length and elements are chosen by the user.

### ***Problem 4***

Modify the findkey program in section 8.4.1 on page 85 to find every occurrence of the desired key instead of only finding the first occurrence.

### ***Problem 5***

Modify the findkey program in section 8.4.1 on page 85 save all occurrences of the key as a list and then display the list.

### ***Problem 6***

Write a subroutine to imitate the effect of .+ for matrices. Your subroutine should take two matrices and produce the matrix whose elements are the sums of the corresponding elements in the two original matrices.

### ***Problem 7***

Modify the chuzcus program on page 96 to search for entries by other fields. That is, expand the program in such a way that you can look up entries by address, telephone number, or ZIP code as well as by last name. Assume that each entry is unique.

***Problem 8***

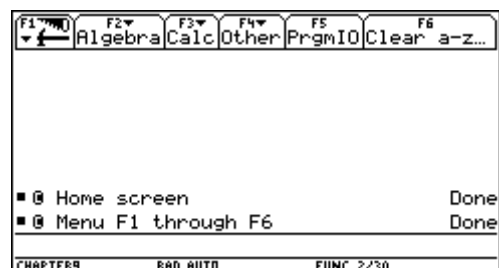
Expand the chuzcus program in Problem 7 in such a way that you can input entries.



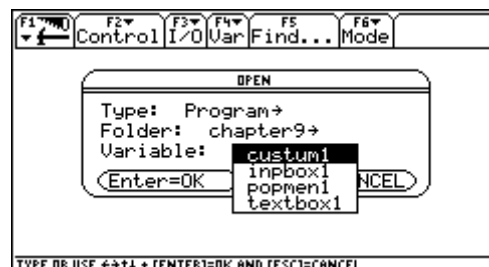
## 9. Menus and dialog boxes

To access many of the features of the TI-92, you use toolbar menus and dialog boxes.

For example, the Home screen provides selections for six menus.



When opening an existing program in the Program Editor, you see a dialog box that allows you to select programs or functions, folder names, and variable names. In this chapter, you will learn how to create your own menus and dialog boxes.



### 9.1 Designing Multiple Menus

Let's look at a program that uses more than one menu to control input.

Suppose that you want to set up an address list on your TI-92. At first glance, this seems fairly easy since you need to enter a name, a street address, a ZIP code and a city. As you look through your address book, however, you begin to realize that many addresses don't follow this pattern. For example, a business address needs an entry for the company name, and a foreign address needs an entry for the country. Therefore, the form of your input will depend on the kind of address you're entering.

The toolbar provides a perfect solution to the problem of differently formatted inputs. You can put entries in the menu corresponding to the different kinds of addresses and then request the appropriate information for the given type of address. Let's assume that you have three types: personal, business, and foreign. Therefore, you want an input subroutine for each type of address.

1. Start a new program in the Program Editor and name it address1
2. Enter the three subroutines shown to the right.
3. Create a toolbar menu to allow the user to select an address format.  
Each item statement corresponds to a label.  
Each label is followed by a statement to call one of the three input subroutines. Enter the program lines shown to the right after the input subroutines.

```

address1()
Prgm
" ***** SUBROUTINE *****
Local persinfo " Personal Address
Define persinfo()=Prgm
Input "Name:",name
Input "Street Address:",addr
Input "Zip code and City:",zccity
EndPrgm
" ***** SUBROUTINE *****
Local businfo " Business Address
Define businfo()=Prgm
Input "Name:",name
Input "Company: ",comp
Input "Street Address:",addr
Input "Zip code and City:",zccity
EndPrgm
" ***** SUBROUTINE *****
Local forinfo " Foreign Address
Define forinfo()=Prgm
Input "Enter Name:",name
Input "Enter Street Address:",addr
Input "Postal Code and City:",pccity
Input "Enter Country:",country
EndPrgm

" ***** MAIN ROUTINE *****

ClrHome
Toolbar " Selecting Format
Title "Address Formats"
Item "Personal",pers
Item "Business",busi
Item "Foreign",forn
EndTBar
Lbl pers
persinfo()
Goto finish
Lbl busi
businfo()
Goto finish
Lbl forn
forinfo()
Lbl finish

EndPrgm

```



4. Not only are there options for the kind of format you want, but within each format there could be choices, such as a department name or a job title in the business address, or a province name in a foreign address. To cover these possibilities, add additional toolbar menus to be displayed when the user selects either the business or foreign address format.
5. Return to the Home screen and run the address1 program.

After you enter a complete address, press **[F5]** to return to the Home screen. You then can run the program again.

Notice that although the Home screen has six menu selections across the top of the screen, the Toolbar command can only create one menu at a time.

In this example, the first Toolbar command created the menu, which allowed you to choose among personal, business, and foreign addresses. Then, once you selected business, for example, that menu vanished and was replaced by the Additional Information menu.

Note: You cannot nest Toolbar commands to keep multiple menus on the screen simultaneously.

```
.. ***** MAIN ROUTINE *****
```

```
Toolbar " Selecting Format
Title "Address Formats"
Item "Personal",pers
Item "Business",busi
Item "Foreign",for
EndTBar
Lbl pers
persinfo()
Goto finish
Lbl busi
businfo()
Toolbar
Title "Additional Information"
Item "Department Name",dept
Item "Job Title",jobt
Item "None",finish
EndTBar
Lbl dept
Input "Department Name:",deptname
Goto finish
Lbl jobt
Input "Job Title:",jobtitle
Goto finish
Lbl for
forinfo()
Toolbar
Title "Additional Information"
Item "Province",prov
Item "None",finish
EndTBar
Lbl prov
Input "Province Name:",province
Lbl finish
EndPrgm

EndPrgm
```

## 9.2 Creating Dialog Boxes

Another method of communicating with the TI-92 is the dialog box. For example, when you press **[APPS]** 7:Program Editor, and then select 2:Open, the Open dialog box is displayed.

You can create dialog boxes with the Dialog...EndDlog construct. Your dialog box must contain a set of choices, a request for values, or a message.



Commands that define what the dialog box will do are placed between Dialog and EndDialog. The four commands you can use within this construct are:

- Text
- Title
- Request
- DropDown

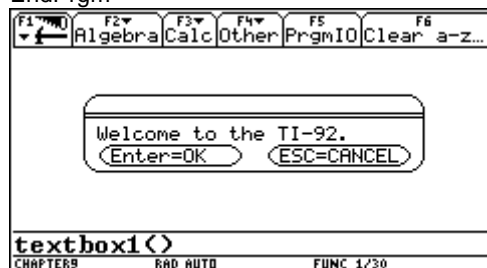
If you try to put any other command within this construction, you'll get a syntax error message.

### The Text Command

The simplest dialog box is one that only contains a message. To display a message, use the Text command. After you read the message, you can press either **ENTER** or **ESC** to close the box.

The short example program `textbox1()` on the right displays an untitled dialog box with the message, "Welcome to the TI-92."

```
textbox1()
Prgm
" Displays Dialog Box with Message
ClrHome
Dialog
Text "Welcome to the TI-92."
EndDialog
EndPrgm
```



### The Title Command

To give a dialog box a title (like the word OPEN at the top of the Open dialog box shown above), use the Title command. However, you cannot use the Title command alone in a dialog box (you'll get a syntax error); a dialog box must contain something in addition to a title.

The example program displays the dialog box from the previous example but this time with the title "Greetings".

```
textbox2()
Prgm
" Displays Titled Dialog Box with Message
Dialog
Title "Greetings"
Text "Welcome to the TI-92."
EndDialog
EndPrgm
```



## The Request Command

Although you can create a dialog box simply to display a message, dialog boxes usually provide a way to enter information, just as the TI-92 uses the Open dialog box to get the name of the program you want to open.

There are two commands to enter information in a dialog box. The first is the Request command. This command displays a prompt followed by an input box. The data you enter in the input box is assigned to a variable. If the variable already has a value, that value will be displayed in the input box initially as a default value.

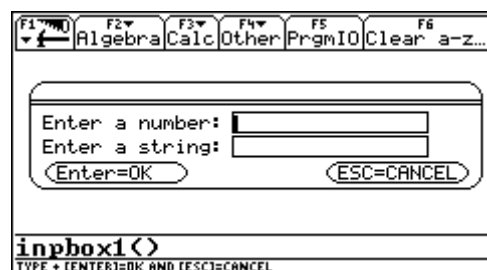
The data you input is interpreted as a string. Therefore, if you type 423 into the input box, the string value "423" will be stored in the variable. To convert this input into a numerical value, use the `expr()` command.

The example program `inpbbox1()` on the right creates two input boxes when executed, one for each Request command.

Run the program.

After the program runs, type each variable name again on the entry line on the Home screen. The values of `bot x` and `str1` will be strings.

```
inpbbox1()
Prgm
" Gets Data Via a Dialog Box
ClrHome
Dialog
Request "Enter a number",x
Request "Enter a string",str1
EndDlog
EndPrgm
```



## The DropDown Command

The DropDown command provides a second way to enter data via a dialog box. This command should be used when you have a limited number of possible values for a variable and you want to ensure that the user only selects one of those possibilities. The DropDown command displays a drop-down menu from which the user can select a value.

Suppose that you want to write a program that will ask the user for the answers to six multiple-choice questions. Each question has four choices, A through D. The example program does this using dialog box drop-down menus.

1. Start a new program in the Program Editor and name it ansbox1.
2. Enter the program lines as shown to the right. It is important to note that the DropDown command returns the number of your selection, not the selection itself. Therefore, the program uses the variable choices to list the letters. Then the proper letter is assigned to anslist; otherwise, anslist would be a list of numbers, no letters.

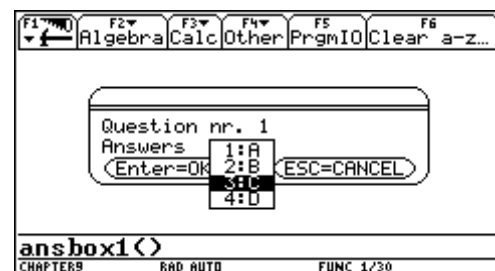
You might try replacing the expression choices[answer] with answer to see what effect it has on the contents of anslist.

3. Run the program.

You will see a dialog box with the message Answers followed by the letter A and an arrow. To choose A without pulling down the menu, press **[ENTER]**. The dialog box for the next question is displayed.

If you want a different answer, press **[▶]** to get the drop-down menu containing the five letter choices A through E. Select a choice and then press **[ENTER][ENTER]** to exit the dialog box and get the dialog box for the next question.

```
ansbox1()
Prgm
" Gets Multiple Choice Responses
ClrHome
{"A","B","C","D"}»choices
newList(7)»anslist
For i,1,7
  Dialog
    Text "Question nr. "string(i)
    DropDown "Answers",choices,answer
  EndDlog
  choices[answer]»anslist[i]
EndFor
Disp anslist
EndPrgm
```



### 9.3 Creating Custom and Pop-Up Menus

The TI-92 provides you two more ways to communicate through menus: Custom menus and Pop-up menus. Custom menus allow you to imitate the toolbar on the Home screen in such a way that you can have multiple menus available at the same time. Pop-up menus are similar to drop-down menus except that they are not displayed in dialog boxes.

#### Custom Menus

Although the Toolbar command is flexible and allows you to do almost anything within a menu, it is also restrictive because you can only have

one menu at a time. The Custom command has the exact opposite properties. It is inflexible in terms of what it allows you to do within its menu, but it is flexible in allowing you to have multiple menus available at the same time.

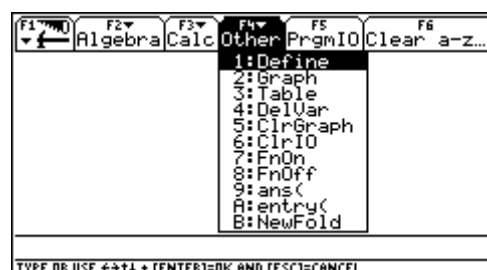
With the Custom command, you specify titles and items as you do with the Toolbar command. The Custom command is different, however, in two respects.

- You can have several titles with each title representing a new menu.
- You do not use labels in the items, only strings. (Choosing an item does not send you to another part of the program but simply selects the string associated with the given item.)

Therefore, the Custom command allows you to create your own menus of TI-92 commands.

Let's look at a simple example. The Home screen toolbar includes a menu entitled  $[F4]$ :Other, which contains a list of many of the commands.

Suppose you are writing a program that only uses a few of these commands but uses them often. With the Custom command, you can set up separate menus specifying only the commands you need.



1. Start a new program in the Program Editor and name it `custom1`.
2. Enter the program lines as shown to the right. Each title represents the title of a new menu. The strings associated with the items below a given title are the commands you can access through that menu.

3. Run the program.

Done appears on the right of the Home screen.

4. To display the new set of menus, press **2nd****[CUSTOM]**. The toolbar changes from the usual six menus to only three menus labeled Expressions, Trig, and Fractions.

Press **[F3]** to pull down the Fractions submenu. Then select 2:getDenom( (getDenom( appears on the entry line of the Home screen), ready to accept an input, just like when you use the standard menus.

When you use the Custom command, your items should be strings that you would want to enter on the entry line.

**Note:** To return to the standard toolbar menus, press [2nd][CUSTOM] again.

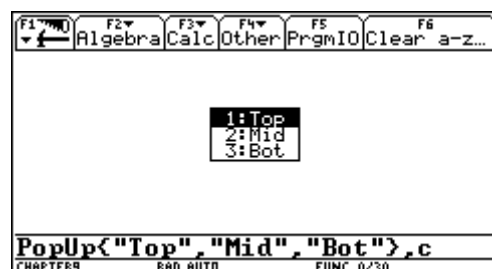
## Pop-Up Menus

The last type of menu command, `PopUp`, creates a pop-up menu. Although pop-up menus are similar to drop-down menus, there are two differences:

- Pop-up menus do not have titles.
- Pop-up menus cannot appear in dialog boxes.

When you enter the `PopUp` command, you specify the choices that will be listed and a variable name to which the selected choice will be assigned. If the variable does not have a value when the pop-up menu is displayed, the first item in the menu is highlighted.

```
custom1()
Prgm
  `` Experiment with Custom Menus
Custom
  Title "Expressions"
  Item "factor("
  Item "expand("
  Title "Trig"
  Item "tExpand("
  Item "tCollect("
  Title "Fractions"
  Item "getNum("
  Item "getDenom("
  Item "propFrac("
EndCustm
EndPrgm
```



If the variable already has a value, which is one of the choices, that value will be highlighted as the default value.

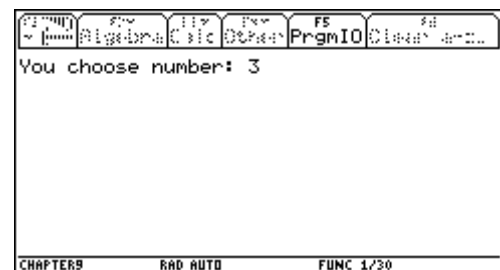
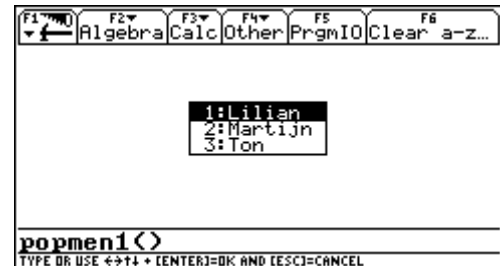
By entering the program `opmen1` on the right, you can experiment with pop-up menus.

```
opmen1()
" Pop-Up Menu example
ClrHome
PopUp {"Lilian","Martijn","Ton"},c
Clrlo
Disp "You choose number: "&string(c)
EndPrgm
```

Select 3: Ton

Press `[ESC]` to return to the Home screen.  
Store `2 → c`, and run the program again.

At last Martijn is highlighted.



## 9.4 Summary of Commands

### Custom

*block of statements*

### EndCustm

Sets up a toolbar that is activated when you press  $\boxed{2nd}[CUSTOM]$ . It is very similar to the ToolBar instruction (page 136) except that Title and Item statements cannot have labels.

**Note:**  $\boxed{2nd}[CUSTOM]$  acts as a toggle. The first instance invokes the menu, and the second instance removes the menu. The menu is removed also when you change applications.

### Dialog

*block of statements*

### Endlog

Generates a dialog box when the program is executed.

Valid *block* options in the  $\boxed{F3}$  I/O , 1: Dialog menu item in the Program Editor are 1: Text , 2: Request, 4: DropDown , and 7: Title .

The variables in a dialog box can be given values that will be displayed as the default (or initial) value. If  $\boxed{ENTER}$  is pressed, the variables are updated from the dialog box and variable ok is set to 1. If  $\boxed{ESC}$  is pressed, its variables are not updated, and system variable ok is set to zero.

### DropDown *titleString, {item1String, item2String, ...,}*

Displays a drop- down menu with the name titleString and containing the items 1: *item1String* , 2: *item2String* , and so forth. DropDown must be within a Dialog... EndDlog block.

If varName already exists and has a value within the range of items, the referenced item is displayed as the default selection. Otherwise, the menu's first item is the default selection.

When you select an item from the menu, the corresponding number of the item is stored in the variable varName . (If necessary, DropDown creates varName.)

### PopUp *itemList, var*

Displays a pop- up menu containing the character strings from *itemList* , waits for you to select an item, and stores the number of your selection in *var* .

The elements of *itemList* must be character strings: *item1String*, *item2String*, *item3String*, ...

If *var* already exists and has a valid item number, that item is displayed as the default choice.





**EndTBar**

Creates a toolbar menu.

The statements can be either Title or Item.

Items must have labels. A Title must also have a label if it does not have an item.

## 9.5 Practical problems

Try the following exercises. Be sure to write a complete function or program using top-down program design, functions and subroutines where needed.

### ***Problem 1***

Modify the address program to add a fourth option, Small, which requires only the name with city and ZIP code (no street address).

### ***Problem 2***

Write a program that will ask the user for a first name, middle and a last name. Then use the toolbar to let the user decide whether to print out the first name followed by middle and last name or last name followed by a comma followed by the first name and the middle name.

### ***Problem 3***

Design a toolbar to allow the user to select among 7-digit, 10-digit, or international telephone numbers. For a 7-digit number, just input the number itself; for a 10-digit number, input the area code and the local number; and for international numbers, input the country code, create another menu to select between the possibility of a city code or no city code, and then input the local number.

### ***Problem 4***

Write a short program to display a dialog box with a message.

- a) Create a dialog box entitled "Ask me" to display the message "Display message"
- b) Modify the program in part (a) to ask the user for the message to be displayed.

### ***Problem 5***

Write a program that uses the dialog box to get two numerical expressions and then prints out their sum. Remember that the input will be interpreted as strings, so you'll need to convert the inputs into numerical expressions before you add them.

***Problem 6***

Use a dialog box to get today's day and date from the user. Use an input box to get the month and date, and use drop-down menus to get the year (assume the year is 1996 through 2000) and the day of the week. Then print out the day and date in a reasonable format.

***Problem 7***

Create a custom menu that rearranges the Other menu on the Home screen. Put the first three commands into a Create menu, the next three commands into a Clear menu, and the next two commands into a Function menu. Use your custom menus to try each of the given commands.

***Problem 8***

Create a custom menu that allows you to run different programs directly from the menus instead of locating them through the Program Editor. If you have the programs from this book on your calculator, group them by chapter in such a way that you can run any program from this chapter by pressing  $\boxed{\text{F8}}$  and then selecting the name of the program you want to run.

***Problem 9***

Convert the drop-down menus from Problem 6 into pop-up menus. Remember that you won't be able to use them within a dialog box.

## Appendix



## TI-92 Functions, Instructions and Commands



Deletes the character to the left of the cursor.



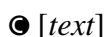
Copies highlighted characters.



Pastes highlighted characters.



Cuts highlighted characters.



press  $\boxed{2nd} X$  A comment symbol  $\bullet$  lets you enter a remark in a program.  
When you run the program, all characters following  $\bullet$  are ignored; comments and are not executed.



press  $\boxed{2nd} H string1 \& string2 \Rightarrow string$   
The  $\&$  symbol stands for appends (concatenates) two strings into one string.

For example: "Ton" & "and" & "Martijn"  $\boxed{ENTER}$  returns: "Ton and Martijn"



**a  $\boxed{STO\triangleright}$  b** Stores the value on the left (a) in the variable on the right (b):



exponent <sup>2</sup>  
To type exponent <sup>2</sup> Press  $\boxed{2nd} \boxed{[CHAR]2:Math\triangleright I:}^2$



"greater than or equal to" symbol  
To type  $\geq$  press  $\boxed{2nd} \boxed{>}\boxed{=}$  or  $\boxed{\blacklozenge} \boxed{.}$  or press  $\boxed{2nd} \boxed{[MATH]}$ , select 8:Test, and select 3:  $\geq$ .



"smaller than or equal to" symbol  
To type  $\leq$  press  $\boxed{2nd} \boxed{<}\boxed{=}$  or  $\boxed{\blacklozenge} \boxed{0}$  or press  $\boxed{2nd} \boxed{[MATH]}$ , select 8:Test, and select 4:  $\leq$ .



"not equal" symbol  
To type  $\neq$  press  $\boxed{\div}\boxed{=}$  or  $\boxed{2nd} \boxed{V}$  or press  $\boxed{2nd} \boxed{[MATH]}$ , select 8:Test, and select 6:  $\neq$ .



*Boolean expression1 and Boolean expression2  $\Rightarrow$  Boolean expression*

Returns true only if both expressions simplify to true. Returns false if either or both expressions evaluate to false.

Returns true or false or a simplified form of the original entry.

For example: true **and** true  $\boxed{ENTER}$  returns: true  
false **and** true  $\boxed{ENTER}$  returns: false

false **and** false **[ENTER]** returns: false

$x \geq 6$  **and**  $x \geq 7$  **[ENTER]** returns:  $x \geq 7$

**approx**(*expression*)

$\Rightarrow$  value

Returns the evaluation of expression as a decimal value, when possible, regardless of the current Exact/ Approx mode.

For example: approx(**[ENTER]**) returns: 3.141 ...

This is equivalent to entering expression and pressing **[♦]** **[ENTER]** on the Home screen.

**char**(*integer*)

$\Rightarrow$  character

Returns a character string containing the character numbered integer from the TI-92 character set. See Appendix B (page 139) for a complete listing of TI-92 characters and their codes.

For example: char(65) **[ENTER]** returns: "A"

char(38) **[ENTER]** returns: "&"

The valid range for *integer* is 0–255.

**Circle** *x, y, r [,drawMode]*

Draws a circle with its center at window coordinates (x, y) and with a radius of r.

x, y, and r must be real values.

If drawMode = 1, draws the circle (default).

If drawMode = 0, turns off the circle.

If drawMode = -1, inverts pixels along the circle.

**ClrDraw**

Clears the Graph screen and resets the Smart Graph feature in such a way that the next time the Graph screen is displayed, the graph will be redrawn.

While viewing the Graph screen, you can clear all drawn items (such as lines and points) by pressing **[F4]** (ReGraph) or pressing **[F6]** and selecting 1:ClrDraw.

**ClrGraph**

Clears any functions or expressions that were graphed with the Graph command or were created with the Table command. (See Graph on page 127)

Any previously selected Y= functions will be graphed the next time that the graph is displayed.

**ClrHome**

Clears all items stored in the **entry()** and **ans()** Home screen history area.

Does not clear the current entry line.

While viewing the Home screen, you can clear the history area by pressing **[F1]** and selecting 8: Clear Home .

**ClrIO**

Clears the Program I/O screen.

**colDim**(*matrix*)

$\Rightarrow$  *expression*

Returns the number of columns contained in matrix.



For example: colDim([0,1,2;3,4,5]) [ENTER] returns: 3

**Note:** See also **rowDim()** on page 135.

**colNorm**(*matrix*)

$\Rightarrow$  *expression*

Returns the maximum of the sums of the absolute values of the elements in the columns in *matrix*.

For example: [0,1,2;3,4,5]»mat [ENTER] returns:  $\begin{bmatrix} 1 & -2 & 3 \\ 4 & 5 & -6 \end{bmatrix}$

colNorm(mat) [ENTER] returns: 9

**Custom**

*block of statements*

**EndCustm**

Sets up a toolbar that is activated when you press [2nd][CUSTOM]. It is very similar to the ToolBar instruction (page 136) except that Title and Item statements cannot have labels.

**Note:** [2nd][CUSTOM] acts as a toggle. The first instance invokes the menu, and the second instance removes the menu. The menu is removed also when you change applications.

**Define** *funcName*(*arg1, arg2,...*) = *expression*

Creates *funcName* as a user- defined function. You then can use *funcName* ( ) , just as you use built- in functions. The function evaluates *expression* using the supplied arguments *arg1, arg2, ...* and returns the result.

*funcName* cannot be the name of a system variable or built- in function.

Note : This command also can be used to define simple variables; for example, **Define** *a* = 3.

**Define** *funcName*(*arg1, arg2,.*) = **Func**

*block of statements*

**EndFunc**

Is identical to the previous form of **Define** except that in this form, the user- defined function *funcName*( ) can execute a block of multiple statements.

*block* also can include expressions and instructions (such as **If...Then...Else**, and **For** ). This allows the function *funcName*( ) to use the **Return** instruction to return a specific result.

Note: It is usually easier to author and edit this form of Function in the program editor rather than on the entry line.

For example: Define absvalue(arg)= Func  
 .. Calculates the absolute value of arg  
 .. Check if  $\arg \geq 0$   
   If  $x \geq 0$  Then  
     Return arg  
   Else

Return -arg  
 EndIf  
 EndFunc

absvalue(-7) **[ENTER]** returns: 7

**Define** *progName*(arg1, arg2,..) = **Prgm**

*block of statements*

**EndPrgm**

Creates *progName* as a program or subprogram, but cannot return a result Return . Can execute a block of multiple statements. *block* also can include expressions and instructions (such as **If...Then...Else**, and **For** ) without restrictions.

**DelFold** *folderNm1* [,*folderNm2*][,*folderNm3*]

Deletes user- defined folders with the names *folderNm1*, *folderNm2*, etc. An error message is displayed if the folders contain any variables.

**Note:** You cannot delete the main folder.

**DelVar** *var1* [,*var2*][,*var3*]

Deletes the specified variables from memory.

**det**(*squareMatrix* )

⇒ *expression*

Returns the determinant of *squareMatrix*.

*squareMatrix* must be square.

For example: det([a,b;c,d]) **[ENTER]**,  $a \cdot d - b \cdot c$   
 det([1,2;3,4]) **[ENTER]** returns: -2

**diag**(*list*)

⇒ *matrix*

Returns a matrix with the values in the argument list or matrix in its main diagonal.

For example: diag(2,4,6) **[ENTER]** returns: 
$$\begin{bmatrix} 2 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 6 \end{bmatrix}$$

**Dialog**

*block of statements*

**Endlog**

Generates a dialog box when the program is executed.

Valid *block* options in the **[F3]** I/O , 1: Dialog menu item in the Program Editor are 1: Text , 2: Request, 4: DropDown , and 7: Title .

The variables in a dialog box can be given values that will be displayed as the default (or initial) value. If **[ENTER]** is pressed, the variables are updated from the dialog box and variable ok is set to 1. If **[ESC]** is pressed, its variables are not updated, and system variable ok is set to zero.

**dim**(*list*)

⇒ *integer*

Returns the dimension of list.

For example: `dim(0,1,2)` `[ENTER]` returns: 3

**dim**(*matrix*)

$\Rightarrow$  *list*

Returns the dimensions of matrix as a two-element list {rows, columns}.

For example: `dim([1,-1,2;-2,3,5])` `[ENTER]` returns: {2,3}

**dim**(*string*)

$\Rightarrow$  *integer*

Returns the number of characters contained in character string *string*.

For example: `dim("Hello")` `[ENTER]` returns: 5

`dim(" Hello"&" there")` `[ENTER]` returns: 11

**Disp**

Displays the current contents of the Program I/O screen.

**Disp** [*exprOrString1*][,*exprOrString2*]

Displays each expression or character string on a separate line of the Program I/O screen.

For example: `Disp "Hi again"` `[ENTER]` returns: Hi again

`Disp sin( $\surd$ 6)` `[ENTER]` returns: 1/2

If Pretty Print = ON , expressions are displayed in pretty print.

**DispG**

Displays the current contents of the Graph screen.

**DispTbl**

Displays the current contents of the Table screen.

**DrawFunc** *expression*

Draws expression as a function of x, using x as the independent variable.

**Note:** Regraphing erases all drawn items.

**DropDown** *titleString*, {*item1String*, *item2String*, ...}

Displays a drop- down menu with the name *titleString* and containing the items 1: *item1String* , 2: *item2String* , and so forth. DropDown must be within a Dialog... EndDialog block.

If *varName* already exists and has a value within the range of items, the referenced item is displayed as the default selection. Otherwise, the menu's first item is the default selection.

When you select an item from the menu, the corresponding number of the item is stored in the variable *varName* . (If necessary, DropDown creates *varName*.)

**Else**

See **If** ,page 127.

**ElseIf**

**If** *Boolean expression1* **Then**  
*block of statements1*

**ElseIf** *Boolean expression2* **Then**  
*block of statements2*

..

..

**ElseIf** *Boolean expressionN* **Then**  
*block of statementsN*

**EndIf**

ElseIf can be used as a program instruction for program branching.  
 See also **If**, page 127.

**EndCustm** See **Custom**, page 139.

**EndDlog** See **Dialog**, page 122.

**EndFor** See **For**, page 125.

**EndFunc** See **Func**, page 126.

**EndIf** See **If**, page 127.

**EndLoop** See **Loop**, page 130.

**EndPrgm** See **Prgm**, page 133.

**EndTBar** See **ToolBar**, page 136.

**EndWhile** See **While**, page 137.

**Exit** Exits the current **For**, **While**, or **Loop** block.  
 Exit is not allowed outside the three looping structures (**For**, **While**, or **Loop**).

**expr(string)**  $\Rightarrow$  *expression*  
 Returns the character string contained in *string* as an expression and immediately executes it.

For example: `expr("3+4+x^5+2x+x")``[ENTER]` returns:  $x^5+3x+7$

**Fill** *expression, matrixVar*  $\Rightarrow$  *matrix*  
 Replaces each element in variable *matrixVar* with *expression*.  
*matrixVar* must already exist.

For example: `[0,1,2;3,4,5]` »`mat1` `[ENTER]` returns:  $\begin{bmatrix} 1 & -2 & 3 \\ 4 & 5 & -6 \end{bmatrix}$

**Fill** 7, `mat1` `[ENTER]` returns:  $\begin{bmatrix} 7 & 7 & 7 \\ 7 & 7 & 7 \end{bmatrix}$

**Fill** *expression, listVar*  $\Rightarrow$  *list*  
 Replaces each element in variable *listVar* with *expression*.  
*listVar* must already exist.

For example: {0,1,2,3,4,5}»list1 ENTER returns: [0 1 2 3 4 5]  
**Fill** 7, list1ENTER returns: [7 7 7 7 7]

**FnOff** Deselects all Y= functions for the current graphing mode.  
 In split-screen, two-graph mode, FnOff only applies to the active graph.

**FnOff** [1] [, 2] .. [, 99]

For example: **FnOff** 1,3 , deselects y1(x) and y3(x), in function graphing mode.

**FnOn** Selects all Y= functions that are defined for the current graphing mode.  
 In split- screen, two- graph mode, **FnOn** only applies to the active graph.

**FnOn** [1] [, 2] ... [, 99]

Selects the specified Y= functions for the current graphing mode.  
**Note:** In 3D graphing mode, only one function at a time can be selected.

**For** *counter*, *begin\_val*, *end\_val* [, *step\_size* ]  
*block of statements*

**EndFor**

Executes the statements in *block of statements* iteratively for each value of *counter* from *begin\_val* upto (or downto) *end\_val*, in increments (or decrements) of *step\_size*.  
*counter* must not be a system variable.  
*step\_size* can be positive or negative, the default value is 1.

**format**(*expr1* [, *formatString*])  $\Rightarrow$  *string*

Returns expression as a character string based on the format template.

*expr1* must simplify to a number.

*formatString* is a string and must be in the form:

F[n], S[n], E[n], G[n][c],

where [ ] indicate optional portions.

F[n] : Fixed format. n is the number of digits to display after the decimal point

S[n] : Scientific format. n is the number of digits to display after the decimal point.

E[n] : Engineering format. n is the number of digits after the first significant digit. The exponent is adjusted to a multiple of three, and the decimal point is moved to the right by zero, one, or two digits.

G[n][c] : Same as fixed format but also separates digits to the left of the radix into groups of three. c specifies the group separator character and defaults to a comma. If c is a period, the radix will be shown as a comma.

[Rc] : Any of the above specifiers may be suffixed with the Rc radix flag, where c is a single character that specifies what to substitute for the radix point.

For example: `format(1.234567,"f3")``[ENTER]` returns: "1.235"  
`format(1.234567,"s2")``[ENTER]` returns: "1.23-0"  
`format(1.234567,"e3")``[ENTER]` returns: "1.235-0"  
`format(1.234567,"g3")``[ENTER]` returns: "1.235"  
`format(1234.567,"g3")``[ENTER]` returns: "1,234.567"  
`format(1.234567,"g3,r:")``[ENTER]` returns: "1:235"

## Func

*block of statements*

## EndFunc

Format required to define a multi(line)statement function.

## Get var

Retrieves a CBL (Calculator- Based Laboratory) value from the link port and stores it in variable *var*.

## getFold()

⇒ *nameString*

Returns the name of the current folder as a string.

For example: `getFold()` `[ENTER]` returns: "main"

## getKey()

⇒ *integer*

Returns the key code of the key pressed.

For a listing of key codes, see Appendix B (page 139)

## getMode(modeNameString)

⇒ *String*

If the argument is a specific mode name, returns a string containing the current setting for that mode.

For example: `getMode("angle")` `[ENTER]` returns: "RADIAN"

If the argument is "ALL", returns a list string pairs containing the settings of all the modes.

## getMode("ALL")

⇒ *String*

For example: `getMode("ALL")` `[ENTER]` returns:

```
{ "Graph" "FUNCTION" "Display Digits" "FLOAT 6" "Angle"
  "RADIAN" "Exponential Format" "NORMAL" "Complex Format"
  "REAL" "Vector Format" "RECTANGULAR" "Pretty Print" "ON"
  "Split Screen" "FULL" "Split 1 App" "Home" "Split 2 App"
  "Graph" "Number of Graphs" "1" "Graph 2" "FUNCTION" "Split
  Screen Ratio" "1: 1" "Exact/ Approx" "AUTO" }
```

**Note 1:** Your screen may display different mode settings.

**Note 2:** If you want to restore the mode settings later, you must store the `getMode("ALL")` result in a variable, and then use `setMode` (see page 135) to restore the modes.

## getType(var)

⇒ *String*

Returns a string indicating the TI-92 data type of variable *var*.

For example: {1,2,3} → temp **ENTER** returns: {1 2 3}  
 getType(temp) **ENTER** returns: "LIST"

### **Data Type Variable Contents**

"DATA"	Data type
"EXPR"	Expression (includes complex/ arbitrary/ undefined, $\infty$ , $-\infty$ , TRUE, FALSE, $\pi$ , $e$ )
"FIG"	Geometry figure
"FUNC"	Function
"GDB"	Graph Data Base
"LIST"	List
"MAC"	Geometry macro
"MAT"	Matrix
"NONE"	Variable does not exist
"NUM"	Real number
"PIC"	Picture
"PRGM"	Program
"STR"	String
"TEXT"	Text type
"VAR"	Name of another variable

### **Goto** *labelName*

Transfers program control to the label *labelName*.

*labelName* must be defined in the same program using a **Lbl** instruction. (See page 129)

### **Graph** *expression[,var]*

Graphs the requested *expression* / *function* using the current graphing mode.

If you omit an optional *var* argument, **Graph** uses the independent variable of the current graphing mode.

For example: Graph sin(t),t **ENTER**

**Note:** Use **ClrGraph** (page 120) to clear these functions, or go to the Y= Editor to re-enable the system Y= functions.

### **identity**(*expression*)

⇒ *matrix*

For example: identity(3) **ENTER** returns: 
$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$
 : the identity matrix with a

dimension 3

*expression* must evaluate to a positive integer.

### **If** *expression*

If *expression* is true, only the statement following is executed; otherwise, the statement is skipped.

### **If** *expression* **Then**

*block of statements*

### **EndIf**

If *expression* is true, the statements in *block of statements* are executed; otherwise, these statements are not executed.

### **If** *expression* **Then**

*block of statements1*

### **Else**

*block of statements2*

**EndIf**

If *expression* is true, the statements in *block of statements1* are executed; otherwise, the statements in *block of statements2* are executed.

**If** *expression1* **Then**

*block of statements1*

**ElseIf** *expression2* **Then**

*block of statements2*

.

.

.

**ElseIf** *expressionN* **Then**

*block of statementsN*

**EndIf**

If *expression1* is true, the statements in *block of statements1* are executed; otherwise, if *expression2* is true, the statements in *block of statements2* are executed; and so on.

**Input** [*promptString*,] *var*

pauses the program, displays *promptString* on the Program I/O screen, waits for you to enter an expression, and stores the expression in variable *var*.

If you omit *promptString* , "?" is displayed as a prompt.

**InputStr** [*promptString*,] *var*

pauses the program, displays *promptString* on the Program I/O screen, waits for you to enter an expression, and stores the expression in variable *var*.

If you omit *promptString* , "?" is displayed as a prompt.

**Note:** The difference between **Input** and **InputStr** is that **InputStr** always stores the result as a string in such a way that " " are not required.

**inString**(*srcString*,*subString*)  $\Rightarrow$  *integer*

Returns the character position in string *srcString* at which the first occurrence of string *subString* begins.

For example: inString("Ton is there","the")**[ENTER]** returns: 8

**int**(*expression*)  $\Rightarrow$  *integer*

Returns the greatest integer that is less than or equal to the *expression*.

For example: int(-2.7)**[ENTER]** returns: -3

**int**(*list1*)  $\Rightarrow$  *list*

**int**(*matrix1*)  $\Rightarrow$  *matrix*

For a list or matrix, returns the greatest integer of each of the elements.

**intDiv**(*number1*,*number2*)  $\Rightarrow$  *integer*

Returns the signed integer part of *number1* divided by *number2*.

For example: intDiv(-9,2)**[ENTER]** returns: -4



- intDiv**(*list1*, *list2*)  $\Rightarrow$  *list*  
**intDiv**(*matrix1*, *matrix2*)  $\Rightarrow$  *matrix*  
 For lists and matrices returns the signed integer part of argument 1 divided by argument 2 for each element pair.
- iPart**(*number*)  $\Rightarrow$  *integer*  
 Returns the integer part of *number*.  
For example: **iPart**(-7,654) [ENTER] returns: -7
- iPart**(*list1*)  $\Rightarrow$  *list*  
**iPart**(*matrix1*)  $\Rightarrow$  *matrix*  
 For lists and matrices, returns the integer part of each element.  
 See **Custom** example on page 121.
- Item** *itemNameString*  
**Item** *itemNameString*, *label*  
 Valid only within a **Custom...EndCustm** or **ToolBar...EndTBar** block. Sets up a dropdown menu element to let you paste text to the cursor position (**Custom**) or branch to a label (**ToolBar**).  
**Note:** Branching to a label is not allowed within a **Custom** block (page 121).
- Lbl** *labelName*  
 Defines a label with the name *labelName* in the program.  
 You can use a **Goto** *labelName* instruction to transfer program control to the instruction immediately following the label.  
*labelName* must meet the same naming requirements as a variable name.
- left**(*srcString*[, *num*])  $\Rightarrow$  *string*  
 Returns the leftmost *num* characters contained in character string *srcString*.  
For example: **left**("Martijn", 2) [ENTER] returns: "Ma"  
 If you omit *num*, returns all of *srcString*.
- left**(*list1*[, *num*])  $\Rightarrow$  *list*  
 Returns the leftmost *num* elements contained in *list1*.  
For example: **left**({ 1,2,-3,4 }, 3) [ENTER] returns: { 1 2 -3 }  
 If you omit *num*, returns all of *list1*.
- left**(*comparison*)  $\Rightarrow$  *expression*  
 Returns the left- hand side of an equation or inequality.  
For example: **left**( $x < 7$ ) [ENTER] returns:  $x$
- Line** *xStart*, *yStart*, *xEnd*, *yEnd* [, *drawMode*]  
 Displays the Graph screen and draws, erases, or inverts a line segment between the window coordinates (*xStart*, *yStart*) and (*xEnd*, *yEnd*), including both endpoints.  
 If *drawMode* = 1, draws the line (default)  
 If *drawMode* = 0, turns off the line  
 If *drawMode* = -1, turns a line that is on to off or off to on.  
**Note:** Regraphing erases all drawn items.

**Local** *var1[,var2][,var3]*

Declares the specified vars as local variables. Those variables exist only during evaluation of a program or function and are deleted when the program or function finishes execution.

**Note:** Local variables save memory because they only exist temporarily. Also, they do not disturb any existing global variable values. Local variables must be used for For loops and for temporarily saving values in a multiline function since modifications on global variables are not allowed in a function.

**Loop**

*block of statements*

**EndLoop**

Repeatedly executes the statements in *block of statements*.

**Note** that the loop will be executed endlessly, unless a Goto or Exit instruction is executed within *block of statements*.

**max**(expression1,expression2)  $\Rightarrow$  *expression*

Returns the maximum of expression1 and expression2.

For example: max(1.2,3.4) [ENTER] returns: 3.4

**max**(list1,list2)  $\Rightarrow$  *list*

**max**(matrix1,matrix2)  $\Rightarrow$  *matrix*

If the arguments are two lists or matrices, returns a list or matrix containing the maximum value of each pair of corresponding elements.

**mid**(srcString,start[,count])  $\Rightarrow$  *string*

Returns *count* characters from character string *srcString*, beginning with character number *start*.

For example: mid("Hello there",1,5)[ENTER] returns: "Hello"

If *count* is omitted or is greater than the dimension of *srcString*, returns all characters from *srcString*, beginning with character number *start*.

For example: mid("Hello there",2)[ENTER] returns: "ello there"

*count* must be  $\geq 0$ . If *count* = 0, returns an empty string.

**mid**(srcList,start[,count])  $\Rightarrow$  *list*

Returns *count* elements from *srcList*, beginning with element number *start*.

For example: mid({9,8,7,6},2,2)[ENTER] returns: {8 7}

If *count* is omitted or is greater than the dimension of *srcList*, returns all elements from *srcList*, beginning with element number *start*.

For example: mid({9,8,7,6},3)[ENTER] returns: {7 6}

*count* must be  $\geq 0$ . If *count* = 0, returns an empty string.

**min**(expression1,expression2)  $\Rightarrow$  *expression*

Returns the minimum of the two arguments.

For example:  $\text{min}(9.8, 7.6)$  [ENTER] returns: 7.6

**min**(*list1*, *list2*)

$\Rightarrow$  *list*

Returns the minimum of the two arguments. If the arguments are two lists, returns a list containing the minimum value of each pair of corresponding elements.

For example:  $\text{min}(\{9, 6\}, \{7, 8\})$  [ENTER] returns: {7.6}

**min**(*list*)

$\Rightarrow$  expression

Returns the minimum element of *list*.

For example:  $\text{min}(\{9, 6, 7, 8\})$  [ENTER] returns: 6

**Note:** See also **max**() (page 130)

**mod**(*expression1*, *expression2*)  $\Rightarrow$  *expression*

Returns the first argument modulo the second argument as defined by the identities:

$$\text{mod}(x, 0) \equiv x$$

$$\text{mod}(x, y) \equiv x - y \cdot \text{int}(x/y)$$

For example:  $\text{mod}(14, 3)$  [ENTER] returns: 2

$$\text{mod}(14, 3) \equiv 14 - 3 \cdot \text{int}(14/3) = 14 - 3 \cdot 4 = 2$$

**Note:** See also **remain**() on page 133.

**mod**(*list1*, *list2*)

$\Rightarrow$  *list*

**mod**(*matrix1*, *matrix2*)

$\Rightarrow$  *matrix*

If the arguments are two lists or two matrices, returns a list or matrix containing the modulo of each pair of corresponding elements.

**NewData** *dataVar*, *list1* [, *list2*] [, *list3*] ...

Creates data variable *dataVar*, where the columns are the lists in order.

Must have at least one list.

For example: **NewData** mydata, {1, 2, 3}, {4, 5, 6} [ENTER] returns: Done

(Go to the Data/ Matrix Editor and open the var *mydata* (or the current) to display the data variable on the right.) **NewData** makes the new variable current in the Data/ Matrix Editor.

F1	F2	F3	F4	F5	F6	F7
Plot	Setup	Cell	Header	Calc	Util	Stat
DATA	c1	c2	c3	c4	c5	
1	1	4				
2	2	5				
3	3	6				
4						
5						
6						
7						
r1c1=1						
APPEND_R      RAD AUTO      FUNC						

**NewFold** *folderName*

Creates a user-defined folder with the name *folderName*, and then sets the current folder to that folder. After you execute this instruction, you are in the new folder.

**newList**(*numElements*)

$\Rightarrow$  *list*

Returns a list with a dimension of *numElements*. Each element is zero.

For example: newList(7)[ENTER] returns: {0 0 0 0 0 0 0}

**newMat**(numRows,numCols)  $\Rightarrow$  *matrix*

Returns a matrix of zeros with the dimension numRows by numCols.

For example: newMat(7,2)[ENTER] returns:  $\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$

**not**(Boolean expression1)  $\Rightarrow$  *Boolean expression*

Returns true, false, or a simplified Boolean expression1 .

For example: not(2>=3)[ENTER] returns: true

not(x<7)[ENTER] returns: x $\geq$ 7

not(not(trueborn))[ENTER] returns: trueborn

**or** *Boolean expression1 or Boolean expression2*  $\Rightarrow$  *Boolean expression*

Returns true if either or both expressions simplify to true. Returns false only if both expressions evaluate to false.

Returns true or false or a simplified form of the original entry.

For example: true or true[ENTER] returns: true

false or true[ENTER] returns: true

false or false[ENTER] returns: false

x $\geq$ 7 or x $\geq$  8 [ENTER] returns: x $\geq$ 7

**ord**(string)  $\Rightarrow$  *integer*

Returns the numeric code of the first character in character string *string*.

For example: ord("Ton")[ENTER] returns: 84

ord("T")[ENTER] returns: 84

**Note:** ord("t")[ENTER] returns: 116

char(116)[ENTER] returns: "t"

ord(char(116))[ENTER] returns: 116

See Appendix B, page 139, for a complete listing of TI-92 characters and their codes.

**Output** row,column,ExprOrString

Displays *ExprOrString* (an *expression* or *character string*) on the Program I/O screen at the text coordinates (row,column).

**Pause** [expression]

Suspends program execution.

If you include *expression* , displays *expression* on the Program I/O screen.

**PopUp** itemList,var

Displays a pop- up menu containing the character strings from *itemList* , waits for you to select an item, and stores the number of



**Request** *promptString*, *var*

Request creates an input/dialog box for the user to type in data. If *var* contains a string, it is displayed and highlighted in the input/dialog box as a default choice. *promptString* must be  $\leq 20$  characters.

For example: Request "Your Name", str1[ENTER] returns:

**Return**[*exp*]

Returns the expression *exp* as a result of the function. Use within a **Func...EndFunc** block.

For example: Define abs(x)=Func

: If  $x \geq 0$  Then

: Return x

: Else

: Return  $-x$

: EndIf

:EndFunc [ENTER]

abs(-7) [ENTER] returns: 7

**right**(*list1*, *num*)

$\Rightarrow$  *list*

Returns the rightmost *num* elements contained in *list1*

For example: right({1,2,3,4,5,6,7},3)[ENTER] returns: {5 6 7}

**right**(*srcString*, *num*)

$\Rightarrow$  *string*

Returns the rightmost *num* characters contained in character string *srcString*.

For example: right("Hello Ton",2)[ENTER] returns: "on"

**right**(*comparison*)

$\Rightarrow$  *expression*

Returns the right side of an equation or inequality

For example: right( $x < 3$ )[ENTER] returns: 3

**round**(*expression1* [, *digits*])

$\Rightarrow$  *expression*

Returns the argument rounded to the specified number of digits after the decimal point.

*digits* must be an integer in the range 0–12. If *digits* is not included, returns the argument rounded to 12 significant digits.

**Note:** Display digits mode may still affect how this is displayed.

For example: round(0.123456789,7) [ENTER] returns: 0.1234568

**round**(*list1*[,*digits*])  
**round**(*matrix1*[,*digits*])

Returns a list / matrix of the elements rounded to the specified number of digits.

**rowDim**(*matrix*)

⇒ *expression*

Returns the number of rows in *matrix*.

**Note:** See also **colDim**() on page 120.

**rowNorm**(*matrix*)

⇒ *expression*

Returns the maximum of the sums of the absolute values of the elements in the rows in *matrix*.

**Note:** All matrix elements must simplify to numbers.

See also **colNorm**() on page 121.

**setFold**(*newfolderName*)

⇒ *oldfolderString*

Returns the name of the current folder as a (*oldfolder*)string and sets *newfolderName* as the current folder.

**Note:** The folder *newfolderName* must exist.

**setMode**(*modeString*,*setString*) ⇒ *string*

Sets mode *modeString* to the new setting *setString*, and returns the (old) current setting of that mode.

For example: `setMode("Angle","Degree")` [ENTER] returns: "RADIAN"  
`sin(30)` [ENTER] returns: 1/2  
`setMode("Angle","Radian")` [ENTER] returns: "DEGREE"  
`sin( $\pi/6$ )` [ENTER] returns: 1/2

*setString* is a character string that specifies the new setting for the mode. It must be one of the settings listed below for the specific mode you are setting.

**setMode**(*list*)

⇒ *stringList*

*list* contains pairs of keyword strings and will set them all at once. This is recommended for multiple mode changes.

**Note:** Use **setMode**(*var*) to restore settings saved with **getMode**("ALL") → *var*. See **getMode** on page 126.

#### Mode Name      Settings

"Graph"	"Function", "Parametric", "Polar", "Sequence", "3D"
"Display Digits"	"Fix 0", "Fix 1", ..., "Fix 12", "Float", "Float 1", ..., "Float 12"
"Angle"	"Radian", "Degree"
"Exponential Format"	"Normal", "Scientific", "Engineering"
"Complex Format"	"Real", "Rectangular", "Polar"
"Vector Format"	"Rectangular", "Cylindrical", "Spherical"
"Pretty Print"	"Off", "On"
"Split Screen"	"Full", "Top- Bottom", "Left- Right"

```

"Split 1 App"      "Home", "Y= Editor", "Window Editor",
"Graph", "Table",
"Data/ Matrix Editor", "Program Editor", "Geometry",
"Text Editor"
"Split 2 App"      "Home", "Y= Editor", "Window Editor",
"Graph", "Table",
"Data/ Matrix Editor", "Program Editor", "Geometry",
"Text Editor"
"Number of Graphs"  "1", "2"
"Graph2"  "Function", "Parametric", "Polar", "Sequence", "3D"
"Split Screen Ratio"  "1:1", "1:2", "2:1"
"Exact/ Approx"  "Auto", "Exact", "Approximate"

```

**Stop**

Used as a program instruction to stop program execution.

**string(expression)**

⇒ *string*

Simplifies *expression* and returns the result as a character string.

For example: string(0.1234567)[ENTER] returns: "0.1234567"

string(1+2)[ENTER] returns: "3"

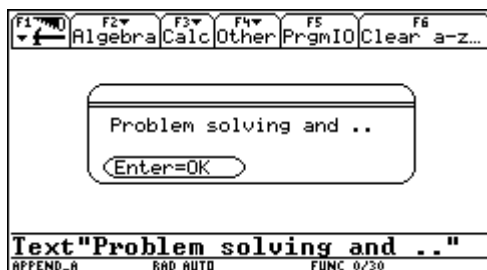
string(sin( $\pi/6$ ) + 7)[ENTER] returns: "15/2"

string(7+45)[ENTER] returns "52". The string "52" represents the characters "5" and "2", not the number 52.

**Text promptString**

Displays the character string *promptString* dialog box.

For example: Text "Problem solving and .."[ENTER] returns:



If used as part of a **Dialog...EndDlog** block, *promptString* is displayed inside that dialog box.

**Title titleString,[Lbl]**

Creates the title of a pull- down menu or dialog box when used inside a **Toolbar** or **Custom** construct, or a **Dialog...EndDlog** block.

**Note:** *Lbl* is only valid in the Toolbar construct. When present, it allows the menu choice to branch to a specified label inside the program.

**Toolbar**

*block of statements*

**EndTBar**

Creates a toolbar menu.

The statements can be either Title or Item.

Items must have labels. A Title must also have a label if it does not have an item.



**Try***block of statements1***Else***block of statements2***EndTry**

Executes *block of statements1* unless an error occurs.  
 Program execution transfers to *block of statements2* if an error occurs in *block of statements1*. Variable *errornum* contains the error number to allow the program to perform error recovery.

**While** *condition**block of statements***EndWhile**

Executes the statements in *block of statements* as long as *condition* is true.

You must allow for the value *condition* to be changed within the *block of statements*.

**zeros**(*expression*, *var*) $\Rightarrow$  *string*

Returns a list of candidate real values of *var* that make *expression* = 0.

For example: zeros( $a \cdot x^2 + b \cdot x + c$ , *x*) [ENTER] returns:

$$\left\{ \frac{-(\sqrt{-(4 \cdot a \cdot c - b^2)} + b)}{2 \cdot a} \quad \frac{(\sqrt{-(4 \cdot a \cdot c - b^2)} - b)}{2 \cdot a} \right\}$$

For some purposes, the result form for zeros() is more convenient than that of solve(). However, the result form of zeros() cannot express implicit solutions, solutions that require inequalities, or solutions that do not involve *var*.

**ZoomStd**

Sets the window variables to the standard values, and then updates the ZoomStd viewing window.

Standard values for function graphing:

*x*:[-10,10,1], *y*:[-10,10,1] and *xres*=2



## TI-92 Character Codes

The **char()** function lets you refer to any TI-92 character by its numeric character code.

For example, to display 2 on the Program I/O screen, use **Disp char(127)**. You can use the **ord()** function to find the numeric code of a character. For example, **ord("A")** returns the value 65.

1	SOH	41	)	81	Q	121	y	161	ı	201	É	241	ñ
2	STX	42	*	82	R	122	z	162	ç	202	Ê	242	ò
3	ETX	43	+	83	S	123	{	163	£	203	Ë	243	ó
4	EOT	44	,	84	T	124		164	¤	204	Ì	244	ô
5	ENQ	45	-	85	U	125	}	165	¥	205	Í	245	õ
6	ACK	46	.	86	V	126	~	166		206	Î	246	ö
7	BELL	47	/	87	W	127	◆	167	§	207	Ï	247	÷
8	BS	48	0	88	X	128	α	168	§	208	Ð	248	ø
9	TAB	49	1	89	Y	129	β	169	●	209	Ñ	249	ù
10	LF	50	2	90	Z	130	Γ	170	ª	210	Ò	250	ú
11	↵	51	3	91	[	131	γ	171	«	211	Ó	251	û
12	FF	52	4	92	\	132	Δ	172	¬	212	Ô	252	ü
13	CR	53	5	93	]	133	δ	173	-	213	Õ	253	ý
14	☐	54	6	94	^	134	ε	174	®	214	Ö	254	þ
15	✓	55	7	95	_	135	ζ	175	-	215	×	255	ÿ
16	▪	56	8	96	`	136	θ	176	◦	216	Ø		
17	◀	57	9	97	a	137	λ	177	+	217	Ù		
18	▶	58	:	98	b	138	ξ	178	²	218	Ú		
19	▲	59	;	99	c	139	Π	179	³	219	Û		
20	▼	60	<	100	d	140	π	180	⁻¹	220	Ü		
21	←	61	=	101	e	141	ρ	181	μ	221	Ý		
22	→	62	>	102	f	142	Σ	182	¶	222	Þ		
23	↑	63	?	103	g	143	σ	183	•	223	ß		
24	↓	64	@	104	h	144	τ	184	×	224	à		
25	◀	65	A	105	i	145	φ	185	´	225	á		
26	▶	66	B	106	j	146	ψ	186	μ	226	â		
27	†	67	C	107	k	147	Ω	187	»	227	ã		
28	∪	68	D	108	l	148	ω	188	¸	228	ä		
29	∩	69	E	109	m	149	Ε	189	·	229	å		
30	⊂	70	F	110	n	150	e	190	,	230	æ		
31	∈	71	G	111	o	151	û	191	¿	231	ç		
32	SPACE	72	H	112	p	152	~	192	À	232	è		
33	!	73	I	113	q	153	τ	193	Á	233	é		
34	"	74	J	114	r	154	¯	194	Â	234	ê		
35	#	75	K	115	s	155	ȳ	195	Ã	235	ë		
36	\$	76	L	116	t	156	œ	196	Ä	236	ì		
37	%	77	M	117	u	157	□	197	Å	237	í		
38	&	78	N	118	v	158	≥	198	Æ	238	î		
39	'	79	O	119	w	159	ÿ	199	Ç	239	ï		
40	(	80	P	120	x	160	..	200	È	240	ð		

