

# **Introduction To Programming With MathRider And MathPiper**

**by Ted Kosan**

Copyright © 2009 by Ted Kosan

This work is licensed under the Creative Commons  
Attribution-ShareAlike 3.0 License. To view a copy of  
this license, visit  
<http://creativecommons.org/licenses/by-sa/3.0/>

## Table of Contents

1	Preface.....	9
1.1	Dedication.....	9
1.2	Acknowledgments.....	9
1.3	Support Email List.....	9
1.4	Recommended Weekly Sequence When Teaching A Class With This Book. .	9
2	Introduction.....	10
2.1	What Is A Mathematics Computing Environment?.....	10
2.2	What Is MathRider?.....	11
2.3	What Inspired The Creation Of Mathrider?.....	12
3	Downloading And Installing MathRider.....	14
3.1	Installing Sun's Java Implementation.....	14
3.1.1	Installing Java On A Windows PC.....	14
3.1.2	Installing Java On A Macintosh.....	14
3.1.3	Installing Java On A Linux PC.....	14
3.2	Downloading And Extracting.....	14
3.2.1	Extracting The Archive File For Windows Users.....	15
3.2.2	Extracting The Archive File For Unix Users.....	15
3.3	MathRider's Directory Structure & Execution Instructions.....	16
3.3.1	Executing MathRider On Windows Systems.....	16
3.3.2	Executing MathRider On Unix Systems.....	17
3.3.2.1	MacOS X.....	17
4	The Graphical User Interface.....	18
4.1	Buffers And Text Areas.....	18
4.2	The Gutter.....	18
4.3	Menus.....	18
4.3.1	File.....	19
4.3.2	Edit.....	19
4.3.3	Search.....	19
4.3.4	Markers, Folding, and View.....	20
4.3.5	Utilities.....	20
4.3.6	Macros.....	20
4.3.7	Plugins.....	20
4.3.8	Help.....	20
4.4	The Toolbar.....	20
4.4.1	Undo And Redo.....	21
5	MathPiper: A Computer Algebra System For Beginners.....	22
5.1	Numeric Vs. Symbolic Computations.....	22

5.2 Using The MathPiper Console As A Numeric (Scientific) Calculator.....	23
5.2.1 Functions.....	24
5.2.1.1 The Sqrt() Square Root Function.....	24
5.2.1.2 The IsEven() Function.....	25
5.2.2 Accessing Previous Input And Results.....	26
5.3 Saving And Restoring A Console Session.....	26
5.3.1 Syntax Errors.....	26
5.4 Using The MathPiper Console As A Symbolic Calculator.....	27
5.4.1 Variables.....	27
5.4.1.1 Calculating With Unbound Variables.....	28
5.4.1.2 Variable And Function Names Are Case Sensitive.....	30
5.4.1.3 Using More Than One Variable.....	30
5.5 Exercises.....	31
5.5.1 Exercise 1.....	31
5.5.2 Exercise 2.....	31
5.5.3 Exercise 3.....	31
5.5.4 Exercise 4.....	32
5.5.5 Exercise 5.....	32
6 The MathPiper Documentation Plugin.....	33
6.1 Function List.....	33
6.2 Mini Web Browser Interface.....	33
6.3 Exercises.....	34
6.3.1 Exercise 1.....	34
6.3.2 Exercise 2.....	34
7 Using MathRider As A Programmer's Text Editor.....	35
7.1 Creating, Opening, Saving, And Closing Text Files.....	35
7.2 Editing Files.....	35
7.3 File Modes.....	35
7.4 Learning How To Type Properly Is An Excellent Investment Of Your Time.....	36
7.5 Exercises.....	36
7.5.1 Exercise 1.....	36
8 MathRider Worksheet Files.....	37
8.1 Code Folds.....	37
8.1.1 The title Attribute.....	38
8.2 Automatically Inserting Folds & Removing Unpreserved Folds.....	38
8.3 Placing Text Outside Of A Fold.....	39
8.4 Exercises.....	39
8.4.1 Exercise 1.....	40
8.4.2 Exercise 2.....	40
8.4.3 Exercise 3.....	40
8.4.4 Exercise 4.....	40

9 MathPiper Programming Fundamentals.....	41
9.1 Values and Expressions.....	41
9.2 Operators.....	41
9.3 Operator Precedence.....	42
9.4 Changing The Order Of Operations In An Expression.....	43
9.5 Functions & Function Names.....	44
9.6 Functions That Produce Side Effects.....	45
9.6.1 Printing Related Functions: Echo(), Write(), And Newline().....	45
9.6.1.1 Echo().....	45
9.6.1.2 Echo Functions Are Useful For "Debugging" Programs.....	47
9.6.1.3 Write().....	48
9.6.1.4 NewLine().....	48
9.7 Expressions Are Separated By Semicolons.....	49
9.7.1 Placing More Than One Expression On A Line In A Fold.....	49
9.7.2 Placing Multiple Expressions In A Code Block.....	50
9.7.2.1 Automatic Bracket, Parentheses, And Brace Match Indicating....	51
9.8 Strings.....	52
9.8.1 The MathPiper Console and MathPiper Folds Can Access The Same Variables.....	52
9.8.2 Using Strings To Make Echo's Output Easier To Read.....	53
9.8.2.1 Combining Strings With The : Operator.....	53
9.8.2.2 WriteString().....	53
9.8.2.3 Nl().....	54
9.8.2.4 Space().....	54
9.8.3 Accessing The Individual Letters In A String.....	54
9.8.3.1 Indexing Before The Beginning Of A String Or Past The End Of A String.....	55
9.9 Comments.....	56
9.10 How To Tell If MathPiper Has Crashed And What To Do If It Has.....	57
9.11 Exercises.....	57
9.11.1 Exercise 1.....	58
9.11.2 Exercise 2.....	58
9.11.3 Exercise 3.....	58
9.11.4 Exercise 4.....	58
9.11.5 Exercise 5.....	59
9.11.6 Exercise 6.....	59
9.11.7 Exercise 7.....	59
10 Rectangular Selection Mode And Text Area Splitting.....	61
10.1 Rectangular Selection Mode.....	61
10.2 Text area splitting.....	61
10.3 Exercises.....	61
10.3.1 Exercise 1.....	62

11 Working With Random Integers.....	63
11.1 Obtaining Random Integers With The RandomInteger() Function.....	63
11.2 Simulating The Rolling Of Dice.....	64
11.3 Exercises.....	65
11.3.1 Exercise 1.....	65
12 Making Decisions.....	66
12.1 Conditional Operators.....	66
12.2 Predicate Expressions.....	69
12.3 Exercises.....	69
12.3.1 Exercise 1.....	69
12.3.2 Exercise 2.....	70
12.3.3 Exercise 3.....	70
12.4 Making Decisions With The If() Function & Predicate Expressions.....	70
12.4.1 If() Functions Which Include An "Else" Parameter.....	72
12.5 Exercises.....	72
12.5.1 Exercise 1.....	73
12.5.2 Exercise 2.....	73
12.6 The And(), Or(), & Not() Boolean Functions & Infix Notation.....	73
12.6.1 And().....	73
12.6.2 Or().....	75
12.6.3 Not() & Prefix Notation.....	76
12.7 Exercises.....	77
12.7.1 Exercise 1.....	78
12.7.2 Exercise 2.....	78
12.7.3 Exercise 3.....	78
13 The While() Looping Function & Bodied Notation.....	80
13.1 Printing The Integers From 1 to 10.....	80
13.2 Printing The Integers From 1 to 100.....	82
13.3 Printing The Odd Integers From 1 To 99.....	82
13.4 Printing The Integers From 1 To 100 In Reverse Order.....	83
13.5 Expressions Inside Of Code Blocks Are Indented.....	84
13.6 Long-Running Loops, Infinite Loops, & Interrupting Execution.....	84
13.7 A Program That Simulates Rolling Two Dice 50 Times.....	85
13.8 Exercises.....	87
13.8.1 Exercise 1.....	88
13.8.2 Exercise 2.....	88
13.8.3 Exercise 3.....	88
13.8.4 Exercise 4.....	88
14 Predicate Functions.....	89
14.1 Finding Prime Numbers With A Loop.....	90
14.2 Finding The Length Of A String With The Length() Function.....	92

14.3 Converting Numbers To Strings With The String() Function.....	93
14.4 Finding Prime Numbers Which End With 7 (And Multi-line Function Calls) .....	93
14.5 Exercises.....	94
14.5.1 Exercise 1.....	95
14.5.2 Exercise 2.....	95
14.5.3 Exercise 3.....	95
15 Lists: Values That Hold Sequences Of Expressions.....	96
15.1 Append() & Nondestructive List Operations.....	97
15.2 Using While Loops With Lists .....	98
15.2.1 Using A While Loop And Append() To Place Values In A List.....	99
15.3 Exercises.....	100
15.3.1 Exercise 1.....	101
15.3.2 Exercise 2.....	101
15.3.3 Exercise 3.....	101
15.3.4 Exercise 4.....	101
15.4 The ForEach() Looping Function.....	102
15.5 Print All The Values In A List Using A ForEach() function.....	102
15.6 Calculate The Sum Of The Numbers In A List Using ForEach().....	103
15.7 The .. Range Operator.....	104
15.8 Using ForEach() With The Range Operator To Print The Prime Numbers Between 1 And 100.....	104
15.8.1 Using ForEach() And The Range Operator To Place The Prime Numbers Between 1 And 50 Into A List.....	105
15.8.2 Exercises.....	106
15.8.3 Exercise 1.....	106
15.8.4 Exercise 2.....	107
15.8.5 Exercise 3.....	107
15.8.6 Exercise 4.....	107
15.8.7 Exercise 5.....	107
16 Functions & Operators Which Loop Internally.....	108
16.1 Functions & Operators Which Loop Internally To Process Lists.....	108
16.1.1 TableForm().....	108
16.1.2 Contains().....	108
16.1.3 Find().....	109
16.1.4 Count().....	109
16.1.5 Select().....	110
16.1.6 The Nth() Function & The [] Operator.....	110
16.1.7 The : Prepend Operator.....	111
16.1.8 Concat().....	111
16.1.9 Insert(), Delete(), & Replace().....	111
16.1.10 Take() .....	112

16.1.11 Drop()	113
16.1.12 FillList()	113
16.1.13 RemoveDuplicates()	114
16.1.14 Reverse()	114
16.1.15 Partition()	114
16.1.16 Table()	115
16.1.17 HeapSort()	116
16.2 Functions That Work With Integers	116
16.2.1 RandomIntegerVector()	116
16.2.2 Max() & Min()	116
16.2.3 Div() & Mod()	117
16.2.4 Gcd()	118
16.2.5 Lcm()	118
16.2.6 Sum()	119
16.2.7 Product()	119
16.3 Exercises	119
16.3.1 Exercise 1	120
16.3.2 Exercise 2	120
16.3.3 Exercise 3	120
16.3.4 Exercise 4	120
16.3.5 Exercise 5	120
16.3.6 Exercise 6	120
17 Nested Loops	121
17.1 Generate All The Combinations That Can Be Entered Into A Two Digit Wheel Lock Using A Nested Loop	121
17.2 Exercises	122
17.2.1 Exercise 1	123
17.2.2 Exercise 2	123
18 User Defined Functions	124
18.1 Global Variables, Local Variables, & Local()	126
18.2 Exercises	128
18.2.1 Exercise 1	128
18.2.2 Exercise 2	128
18.2.3 Exercise 3	128
19 Miscellaneous topics	129
19.1 Incrementing And Decrementing Variables With The ++ And -- Operators	129
19.1.1 Incrementing Variables With The ++ Operator	129
19.1.2 Decrementing Variables With The -- Operator	130
19.2 Exercises	131
19.2.1 Exercise 1	131





# 1 Preface

## 2 1.1 Dedication

3 This book is dedicated to Steve Yegge and his blog entry "Math Every Day"  
4 (<http://steve.yegge.googlepages.com/math-every-day>).

## 5 1.2 Acknowledgments

6 The following people have provided feedback on this book (if I forgot to include  
7 your name on this list, please email me at ted.kosan at gmail.com):

8 Susan Addington

9 Matthew Moelter

10 Sherm Ostrowsky

## 11 1.3 Support Email List

12 The support email list for this book is called **mathrider-**  
13 **users@googlegroups.com** and you can subscribe to it at  
14 <http://groups.google.com/group/mathrider-users>.

## 15 1.4 Recommended Weekly Sequence When Teaching A Class With This 16 Book

- 17 • Week 1: Sections 1 - 6.
- 18 • Week 2: Sections 7 - 9.
- 19 • Week 3: Sections 10 - 13.
- 20 • Week 4: Sections 14 - 15.
- 21 • Week 5: Sections 16 - 19.

## 22 **2 Introduction**

23 MathRider is an open source mathematics computing environment for  
24 performing numeric and symbolic computations (the difference between numeric  
25 and symbolic computations are discussed in a later section). Mathematics  
26 computing environments are complex and it takes a significant amount of time  
27 and effort to become proficient at using one. The amount of power that these  
28 environments make available to a user, however, is well worth the effort needed  
29 to learn one. It will take a beginner a while to become an expert at using  
30 MathRider, but fortunately one does not need to be a MathRider expert in order  
31 to begin using it to solve problems.

### 32 **2.1 What Is A Mathematics Computing Environment?**

33 A Mathematics Computing Environment is a set of computer programs that 1)  
34 automatically execute a wide range of numeric and symbolic mathematics  
35 calculation algorithms and 2) provide a user interface which enables the user to  
36 access these calculation algorithms and manipulate the mathematical objects  
37 they create (An algorithm is a step-by-step sequence of instructions for solving a  
38 problem and we will be learning about algorithms later in the book).

39 Standard and graphing scientific calculator users interact with these devices  
40 using buttons and a small LCD display. In contrast to this, users interact with  
41 MathRider using a rich graphical user interface which is driven by a computer  
42 keyboard and mouse. Almost any personal computer can be used to run  
43 MathRider, including the latest subnotebook computers.

44 Calculation algorithms exist for many areas of mathematics and new algorithms  
45 are constantly being developed. Software that contains these kind of algorithms  
46 is commonly referred to as "Computer Algebra Systems (CAS)". A significant  
47 number of computer algebra systems have been created since the 1960s and the  
48 following list contains some of the more popular ones:

49 [http://en.wikipedia.org/wiki/Comparison\\_of\\_computer\\_algebra\\_systems](http://en.wikipedia.org/wiki/Comparison_of_computer_algebra_systems)

50 Some environments are highly specialized and some are general purpose. Some  
51 allow mathematics to be entered and displayed in traditional form (which is what  
52 is found in most math textbooks). Some are able to display traditional form  
53 mathematics but need to have it input as text and some are only able to have  
54 mathematics displayed and entered as text.

55 As an example of the difference between traditional mathematics form and text  
56 form, here is a formula which is displayed in traditional form:

$$a = x^2 + 4hx + \frac{3}{7}$$

57 and here is the same formula in text form:

58 
$$a = x^2 + 4 \cdot h \cdot x + 3/7$$

59 Most computer algebra systems contain a mathematics-oriented programming  
60 language. This allows programs to be developed which have access to the  
61 mathematics algorithms which are included in the system. Some mathematics-  
62 oriented programming languages were created specifically for the system they  
63 work in while others were built on top of an existing programming language.

64 Some mathematics computing environments are proprietary and need to be  
65 purchased while others are open source and available for free. Both kinds of  
66 systems possess similar core capabilities, but they usually differ in other areas.

67 Proprietary systems tend to be more polished than open source systems and they  
68 often have graphical user interfaces that make inputting and manipulating  
69 mathematics in traditional form relatively easy. However, proprietary  
70 environments also have drawbacks. One drawback is that there is always a  
71 chance that the company that owns it may go out of business and this may make  
72 the environment unavailable for further use. Another drawback is that users are  
73 unable to enhance a proprietary environment because the environment's source  
74 code is not made available to users.

75 Some open source computer algebra systems do not have graphical user  
76 interfaces, but their user interfaces are adequate for most purposes and the  
77 environment's source code will always be available to whomever wants it. This  
78 means that people can use the environment for as long as they desire and they  
79 can also enhance it.

## 80 **2.2 What Is MathRider?**

81 MathRider is an open source Mathematics Computing Environment which has  
82 been designed to help people teach themselves the [STEM](#) disciplines (Science,  
83 Technology, Engineering, and Mathematics) in an efficient and holistic way. It  
84 inputs mathematics in textual form and displays it in either textual form or  
85 traditional form.

86 MathRider uses MathPiper as its default computer algebra system, BeanShell as  
87 its main scripting language, jEdit as its framework (hereafter referred to as the  
88 MathRider framework), and Java as its overall implementation language. One  
89 way to determine a person's MathRider expertise is by their knowledge of these  
90 components. (see Table 1)

Level	Knowledge
MathRider Developer	Knows Java, BeanShell, and the MathRider framework at an advanced level. Is able to develop MathRider plugins.
MathRider Customizer	Knows Java, BeanShell, and the MathRider framework at an intermediate level. Is able to develop MathRider macros.
MathRider Expert	Knows MathPiper at an advanced level and is skilled at using most aspects of the MathRider application.
MathRider Novice	Knows MathPiper at an intermediate level, but has only used MathRider for a short while.
MathRider Newbie	Does not know MathPiper but has been exposed to at least one programming language.
Programming Newbie	Does not know how a computer works and has never programmed before but knows how to use a word processor.

*Table 1: MathRider user experience levels.*

91 This book is for MathRider and Programming Newbies. This book will teach you  
 92 enough programming to begin solving problems with MathRider and the  
 93 language that is used is MathPiper. It will help you to become a MathRider  
 94 Novice, but you will need to learn MathPiper from books that are dedicated to it  
 95 before you can become a MathRider Expert.

96 The MathRider project website (<http://mathrider.org>) contains more information  
 97 about MathRider along with other MathRider resources.

## 98 **2.3 What Inspired The Creation Of Mathrider?**

99 Two of MathRider's main inspirations are Scott McNeally's concept of "No child  
 100 held back":

101 [http://weblogs.java.net/blog/turbogeek/archive/2004/09/no\\_child\\_held\\_b\\_1.html](http://weblogs.java.net/blog/turbogeek/archive/2004/09/no_child_held_b_1.html)

102 and Steve Yegge's thoughts on learning mathematics:

103 1) Math is a lot easier to pick up after you know how to program. In fact, if  
 104 you're a halfway decent programmer, you'll find it's almost a snap.

105 2) They teach math all wrong in school. Way, WAY wrong. If you teach  
 106 yourself math the right way, you'll learn faster, remember it longer, and it'll  
 107 be much more valuable to you as a programmer.

108 3) The right way to learn math is breadth-first, not depth-first. You need to  
 109 survey the space, learn the names of things, figure out what's what.

110 <http://steve-yegge.blogspot.com/2006/03/math-for-programmers.html>

111 MathRider is designed to help a person learn mathematics on their own with  
112 little or no assistance from a teacher. It makes learning mathematics easier by  
113 focusing on how to program first and it facilitates a breadth-first approach to  
114 learning mathematics.

## 115 **3 Downloading And Installing MathRider**

### 116 **3.1 *Installing Sun's Java Implementation***

117 MathRider is a Java-based application and therefore a current version of Sun's  
118 Java (at least Java 6) must be installed on your computer before MathRider can  
119 be run.

#### 120 **3.1.1 Installing Java On A Windows PC**

121 Many Windows PCs will already have a current version of Java installed. You can  
122 test to see if you have a current version of Java installed by visiting the following  
123 web site:

124 <http://java.com/>

125 This web page contains a link called "Do I have Java?" which will check your Java  
126 version and tell you how to update it if necessary.

#### 127 **3.1.2 Installing Java On A Macintosh**

128 Macintosh computers have Java pre-installed but you may need to upgrade to a  
129 current version of Java (at least Java 6) before running MathRider. If you need  
130 to update your version of Java, visit the following website:

131 <http://developer.apple.com/java.>

#### 132 **3.1.3 Installing Java On A Linux PC**

133 Locate the Java documentation for your Linux distribution and carefully follow  
134 the instructions provided for installing a Java 6 compatible version of Java on  
135 your system.

### 136 **3.2 *Downloading And Extracting***

137 One of the many benefits of learning MathRider is the programming-related  
138 knowledge one gains about how open source software is developed on the  
139 Internet. An important enabler of open source software development are  
140 websites, such as sourceforge.net (<http://sourceforge.net>) and java.net  
141 (<http://java.net>) which make software development tools available for free to  
142 open source developers.

143 MathRider is hosted at java.net and the URL for the project website is:

144 <http://mathrider.org>

145 MathRider can be obtained by selecting the **download** tab and choosing the  
146 correct download file for your computer. Place the download file on your hard  
147 drive where you want MathRider to be located. **For Windows users, it is**  
148 **recommended that MathRider be placed somewhere on c: drive.**

149 The MathRider download consists of a main directory (or folder) called  
150 **mathrider** which contains a number of directories and files. In order to make  
151 downloading quicker and sharing easier, the mathrider directory (and all of its  
152 contents) have been placed into a single compressed file called an **archive**. For  
153 **Windows** systems, the archive has a **.zip** extension and the archives for **Unix-**  
154 **based** systems have a **.tar.bz2** extension.

155 After an archive has been downloaded onto your computer, the directories and  
156 files it contains must be **extracted** from it. The process of extraction  
157 uncompresses copies of the directories and files that are in the archive and  
158 places them on the hard drive, usually in the same directory as the archive file.  
159 After the extraction process is complete, the archive file will still be present on  
160 your drive along with the extracted **mathrider** directory and its contents.

161 The **archive file** can be easily copied to a CD or USB drive if you would like to  
162 install MathRider on another computer or give it to a friend. **However, don't**  
163 **try to run MathRider from a USB drive because it will not work correctly.**

164 **(Note: If you already have a version of MathRider installed and you want**  
165 **to install a new version in the same directory that holds the old version,**  
166 **you must delete the old version first or move it to a separate directory.)**

### 167 3.2.1 Extracting The Archive File For Windows Users

168 Usually the easiest way for Windows users to extract the MathRider archive file  
169 is to navigate to the folder which contains the archive file (using the Windows  
170 GUI), **right click on the archive file (it should appear as a folder with a**  
171 **vertical zipper on it)**, and select **Extract All...** from the pop up menu.

172 After the extraction process is complete, a new folder called **mathrider** should  
173 be present in the same folder that contains the archive file. **(Note: be careful**  
174 **not to double click on the archive file by mistake when you are trying to**  
175 **open the mathrider folder. The Windows operating system will open the**  
176 **archive just like it opens folders and this can fool you into thinking you**  
177 **are opening the mathrider folder when you are not. You may want to**  
178 **move the archive file to another place on your hard drive after it has**  
179 **been extracted to avoid this potential confusion.)**

### 180 3.2.2 Extracting The Archive File For Unix Users

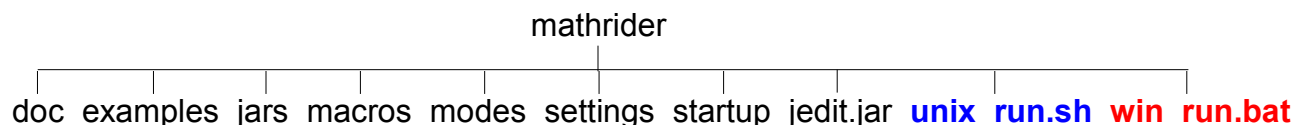
181 One way Unix users can extract the download file is to open a shell, change to  
182 the directory that contains the archive file, and extract it using the following  
183 command:

184     tar -xvjf <name of archive file>

185   If your desktop environment has GUI-based archive extraction tools, you can use  
186   these as an alternative.

### 187   **3.3 MathRider's Directory Structure & Execution Instructions**

188   The top level of MathRider's directory structure is shown in Illustration 1:



*Illustration 1: MathRider's Directory Structure*

189   The following is a brief description this top level directory structure:

190   **doc** - Contains MathRider's documentation files.

191   **examples** - Contains various example programs, some of which are pre-opened  
192   when MathRider is first executed.

193   **jars** - Holds plugins, code libraries, and support scripts.

194   **macros** - Contains various scripts that can be executed by the user.

195   **modes** - Contains files which tell MathRider how to do syntax highlighting for  
196   various file types.

197   **settings** - Contains the application's main settings files.

198   **startup** - Contains startup scripts that are executed each time MathRider  
199   launches.

200   **jedit.jar** - Holds the core jEdit application which MathRider builds upon.

201   **unix\_run.sh** - The script used to execute MathRider on Unix systems.

202   **win\_run.bat** - The batch file used to execute MathRider on Windows systems.

#### 203   **3.3.1 Executing MathRider On Windows Systems**

204   Open the **mathrider** folder **(not the archive file!)** and double click on the  
205   **win\_run** file.



### 206 **3.3.2 Executing MathRider On Unix Systems**

207 Open a shell, change to the **mathrider** folder, and execute the **unix\_run.sh**  
208 script by typing the following:

209 `sh unix_run.sh`

#### 210 **3.3.2.1 MacOS X**

211 Make a note of where you put the Mathrider application (for example  
212 **/Applications/mathrider**). Run Terminal (which is in /Applications/Utilities).  
213 Change to that directory (folder) by typing:

214 `cd /Applications/mathrider`

215 Run mathrider by typing:

216 `sh unix_run.sh`

## 217 4 The Graphical User Interface

218 MathRider is built on top of jEdit (<http://jedit.org>) so it has the "heart" of a  
219 programmer's text editor. Programmer's text editors are similar to standard text  
220 editors (like NotePad and WordPad) and word processors (like MS Word and  
221 OpenOffice) in a number of ways so getting started with MathRider should be  
222 relatively easy for anyone who has used a text editor or a word processor.  
223 However, programmer's text editors are more challenging to use than a standard  
224 text editor or a word processor because programmer's text editors have  
225 capabilities that are far more advanced than these two types of applications.

226 Most software is developed with a programmer's text editor (or environments  
227 which contain one) and so learning how to use a programmer's text editor is one  
228 of the many skills that MathRider provides which can be used in other areas.  
229 The MathRider series of books are designed so that these capabilities are  
230 revealed to the reader over time.

231 In the following sections, the main parts of MathRider's graphical user interface  
232 are briefly covered. Some of these parts are covered in more depth later in the  
233 book and some are covered in other books.

234 **As you read through the following sections, I encourage you to explore**  
235 **each part of MathRider that is being discussed using your own copy of**  
236 **MathRider.**

### 237 4.1 Buffers And Text Areas

238 In MathRider, open files are called **buffers** and they are viewed through one or  
239 more **text areas**. Each text area has a tab at its upper-left corner which displays  
240 the name of the buffer it is working on along with an indicator which shows  
241 whether the buffer has been saved or not. The user is able to select a text area  
242 by clicking its tab and double clicking on the tab will close the text area. Tabs  
243 can also be rearranged by dragging them to a new position with the mouse.

### 244 4.2 The Gutter

245 The gutter is the vertical gray area that is on the left side of the main window. It  
246 can contain line numbers, buffer manipulation controls, and context-dependent  
247 information about the text in the buffer.

### 248 4.3 Menus

249 The main menu bar is at the top of the application and it provides access to a  
250 significant portion of MathRider's capabilities. The commands (or **actions**) in  
251 these menus all exist separately from the menus themselves and they can be  
252 executed in alternate ways (such as keyboard shortcuts). The menu items (and

253 even the menus themselves) can all be customized, but the following sections  
254 describe the default configuration.

### 255 4.3.1 File

256 The File menu contains actions which are typically found in normal text editors  
257 and word processors. The actions to create new files, save files, and open  
258 existing files are all present along with variations on these actions.

259 Actions for opening recent files, configuring the page setup, and printing are  
260 also present.

### 261 4.3.2 Edit

262 The Edit menu also contains actions which are typically found in normal text  
263 editors and word processors (such as **Undo**, **Redo**, **Cut**, **Copy**, and **Paste**).  
264 However, there are also a number of more sophisticated actions available which  
265 are of use to programmers. For beginners, though, the typical actions will be  
266 sufficient for most editing needs.

### 267 4.3.3 Search

268 The actions in the Search menu are used heavily, even by beginners. A good way  
269 to get your mind around the search actions is to open the Search dialog window  
270 by selecting the **Find...** action (which is the first actions in the Search menu). A  
271 **Search And Replace** dialog window will then appear which contains access to  
272 most of the search actions.

273 At the top of this dialog window is a text area labeled **Search for** which allows  
274 the user to enter text they would like to find. Immediately below it is a text area  
275 labeled **Replace with** which is for entering optional text that can be used to  
276 replace text which is found during a search.

277 The column of radio buttons labeled **Search in** allows the user to search in a  
278 **Selection** of text (which is text which has been highlighted), the **Current**  
279 **Buffer** (which is the one that is currently active), **All buffers** (which means all  
280 opened files), or a whole **Directory** of files. The default is for a search to be  
281 conducted in the current buffer and this is the mode that is used most often.

282 The column of check boxes labeled **Settings** allows the user to either **Keep or**  
283 **hide the Search dialog window** after a search is performed, **Ignore the case**  
284 of searched text, use an advanced search technique called a **Regular**  
285 **expression** search (which is covered in another book), and to perform a  
286 **HyperSearch** (which collects multiple search results in a text area).

287 The **Find** button performs a normal find operation. **Replace & Find** will replace  
288 the previously found text with the contents of the **Replace with** text area and  
289 perform another find operation. **Replace All** will find all occurrences of the

290 contents of the **Search for** text area and replace them with the contents of the  
291 **Replace with** text area.

#### 292 **4.3.4 Markers, Folding, and View**

293 These are advanced menus and they are described in later sections.

#### 294 **4.3.5 Utilities**

295 The utilities menu contains a significant number of actions, some that are useful  
296 to beginners and others that are meant for experts. The two actions that are  
297 most useful to beginners are the **Buffer Options** actions and the **Global**  
298 **Options** actions. The **Buffer Options** actions allows the currently selected  
299 buffer to be customized and the **Global Options** actions brings up a rich dialog  
300 window that allows numerous aspects of the MathRider application to be  
301 configured.

302 Feel free to explore these two actions in order to learn more about what they do.

#### 303 **4.3.6 Macros**

304 This is an advanced menu and it is described in a later sections.

#### 305 **4.3.7 Plugins**

306 Plugins are component-like pieces of software that are designed to provide an  
307 application with extended capabilities and they are similar in concept to physical  
308 world components. The tabs on the right side of the application which are  
309 labeled "GeoGebra", "Jung", "MathPiper", "MathPiperDocs", etc. are all plugins  
310 and they can be **opened** and **closed** by clicking on their **tabs**. **Feel free to close**  
311 **any of these plugins which may be opened if you are not currently using**  
312 **them**. MathRider pPlugins are covered in more depth in a later section.

#### 313 **4.3.8 Help**

314 The most important action in the **Help** menu is the **MathRider Help** action.  
315 This action brings up a dialog window with contains documentation for the core  
316 MathRider application along with documentation for each installed plugin.

#### 317 **4.4 The Toolbar**

318 The **Toolbar** is located just beneath the menus near the top of the main window  
319 and it contains a number of icon-based buttons. These buttons allow the user to  
320 access the same actions which are accessible through the menus just by clicking  
321 on them. There is not room on the toolbar for all the actions in the menus to be

322 displayed, but the most common actions are present. The user also has the  
323 option of customizing the toolbar by using the **Utilities->Global Options->Tool**  
324 **Bar** dialog.

#### 325 **4.4.1 Undo And Redo**

326 The **Undo** button on the toolbar is able to undo any text was entered since the  
327 current session of MathRider was launched. This is very handy for undoing  
328 mistakes or getting back text which was deleted. The **Redo** button can be used  
329 if you have selected Undo too many times and you need to "undo" one ore more  
330 Undo operations.

## 331 **5 MathPiper: A Computer Algebra System For Beginners**

332 Computer algebra systems are extremely powerful and very useful for solving  
333 STEM-related problems. In fact, one of the reasons for creating MathRider was  
334 to provide a vehicle for delivering a computer algebra system to as many people  
335 as possible. If you like using a scientific calculator, you should love using a  
336 computer algebra system!

337 At this point you may be asking yourself "if computer algebra systems are so  
338 wonderful, why aren't more people using them?" One reason is that most  
339 computer algebra systems are complex and difficult to learn. Another reason is  
340 that proprietary systems are very expensive and therefore beyond the reach of  
341 most people. Luckily, there are some open source computer algebra systems  
342 that are powerful enough to keep most people engaged for years, and yet simple  
343 enough that even a beginner can start using them. MathPiper (which is based on  
344 a CAS called Yacas) is one of these simpler computer algebra systems and it is  
345 the computer algebra system which is included by default with MathRider.

346 A significant part of this book is devoted to learning MathPiper and a good way  
347 to start is by discussing the difference between numeric and symbolic  
348 computations.

### 349 **5.1 Numeric Vs. Symbolic Computations**

350 A Computer Algebra System (CAS) is software which is capable of performing  
351 both **numeric** and **symbolic** computations. **Numeric** computations are  
352 performed exclusively with numerals and these are the type of computations that  
353 are performed by typical hand-held calculators.

354 **Symbolic** computations (which also called algebraic computations) relate "...to  
355 the use of machines, such as computers, to manipulate mathematical equations  
356 and expressions in symbolic form, as opposed to manipulating the  
357 approximations of specific numerical quantities represented by those symbols."  
358 ([http://en.wikipedia.org/wiki/Symbolic\\_mathematics](http://en.wikipedia.org/wiki/Symbolic_mathematics)).

359 Since most people who read this document will probably be familiar with  
360 performing numeric calculations as done on a scientific calculator, the next  
361 section shows how to use MathPiper as a scientific calculator. The section after  
362 that then shows how to use MathPiper as a symbolic calculator. Both sections  
363 use the console interface to MathPiper. In MathRider, a console interface to any  
364 plugin or application is a text-only **shell** or **command line** interface to it. This  
365 means that you type on the keyboard to send information to the console and it  
366 prints text to send you information.

## 367 **5.2 Using The MathPiper Console As A Numeric (Scientific) Calculator**

368 Open the MathPiper plugin by selecting the **MathPiper** tab in the lower left part  
369 of the MathRider application. The MathPiper **console** interface is a text area  
370 which is inside this plugin. Feel free to increase or decrease the size of the  
371 console text area if you would like by dragging on the dotted lines which are at  
372 the top side and right side of the console window.

373 When the MathPiper console is first launched, it prints a welcome message and  
374 then provides **In>** as an input prompt:

```
375 MathPiper version ".76x".
```

```
376 In>
```

377 Click to the right of the prompt in order to place the cursor there then type **2+2**  
378 followed by **<shift><enter>** (or **<shift><return>** on a Macintosh):

```
379 In> 2+2
```

```
380 Result> 4
```

```
381 In>
```

382 When **<shift><enter>** was pressed, 2+2 was read into MathPiper for  
383 **evaluation** and **Result>** was printed followed by the result **4**. Another input  
384 prompt was then displayed so that further input could be entered. This **input,**  
385 **evaluation, output** process will continue as long as the console is running and  
386 it is sometimes called a **Read, Eval, Print Loop** or **REPL**. In further examples,  
387 the last **In>** prompt will not be shown to save space.

388 In addition to addition, MathPiper can also do subtraction, multiplication,  
389 exponents, and division:

```
390 In> 5-2
```

```
391 Result> 3
```

```
392 In> 3*4
```

```
393 Result> 12
```

```
394 In> 2^3
```

```
395 Result> 8
```

```
396 In> 12/6
```

```
397 Result> 2
```

398 Notice that the multiplication symbol is an asterisk (\*), the exponent symbol is a  
399 caret (^), and the division symbol is a forward slash (/). These symbols (along  
400 with addition (+), subtraction (-), and ones we will talk about later) are called

401 **operators** because they tell MathPiper to perform an operation such as addition  
402 or division.

403 MathPiper can also work with decimal numbers:

```
404 In> .5+1.2  
405 Result> 1.7
```

```
406 In> 3.7-2.6  
407 Result> 1.1
```

```
408 In> 2.2*3.9  
409 Result> 8.58
```

```
410 In> 2.2^3  
411 Result> 10.648
```

```
412 In> 9.5/3.2  
413 Result> 9.5/3.2
```

414 In the last example, MathPiper returned the fraction unevaluated. This  
415 sometimes happens due to MathPiper's symbolic nature, but a result in **numeric**  
416 **form** can be obtained by using the **N()** function:

```
417 In> N(9.5/3.2)  
418 Result> 2.96875
```

419 As can be seen here, when a result is given in numeric form, it means that it is  
420 given as a decimal number. The **N()** function is discussed in the next section.

## 421 5.2.1 Functions

422 **N()** is an example of a **function**. A function can be thought of as a "black box"  
423 which accepts input, processes the input, and returns a result. Each function  
424 has a name and in this case, the name of the function is **N** which stands for  
425 "**numeric**". To the right of a function's name there is always a **set of**  
426 **parentheses** and information that is sent to the function is placed inside of  
427 them. The purpose of the **N()** function is to make sure that the information that  
428 is sent to it is processed numerically instead of symbolically. Functions are used  
429 by **evaluating** them and this happens when <shift><enter> is pressed. Another  
430 name for evaluating a function is **calling** it.

### 431 5.2.1.1 The Sqrt() Square Root Function

432 The following example show the **N()** function being used with the square root  
433 function **Sqrt()**:



```
434 In> Sqrt(9)
```

```
435 Result: 3
```

```
436 In> Sqrt(8)
```

```
437 Result: Sqrt(8)
```

```
438 In> N(Sqrt(8))
```

```
439 Result: 2.828427125
```

440 Notice that Sqrt(9) returned 3 as expected but Sqrt(8) returned Sqrt(8). We  
441 needed to use the N() function to force the square root function to return a  
442 numeric result. The reason that Sqrt(8) does not appear to have done anything  
443 is because computer algebra systems like to work with expressions that are as  
444 exact as possible. In this case the **symbolic** value Sqrt(8) represents the number  
445 that is the square root of 8 more accurately than any decimal number can.

446 For example, the following four decimal numbers all represent  $\sqrt{8}$ , but none of  
447 them represent it more accurately than Sqrt(8) does:

```
448 2.828427125
```

```
449 2.82842712474619
```

```
450 2.82842712474619009760337744842
```

```
451 2.8284271247461900976033774484193961571393437507539
```

452 Whenever MathPiper returns a symbolic result and a numeric result is desired,  
453 simply use the N() function to obtain one. The ability to work with symbolic  
454 values are one of the things that make computer algebra systems so powerful  
455 and they are discussed in more depth in later sections.

#### 456 **5.2.1.2 The IsEven() Function**

457 Another often used function is **IsEven()**. The **IsEven()** function takes a number  
458 as input and returns **True** if the number is even and **False** if it is not even:

```
459 In> IsEven(4)
```

```
460 Result> True
```

```
461 In> IsEven(5)
```

```
462 Result> False
```

463 MathPiper has a large number of functions some of which are described in more  
464 depth in the MathPiper Documentation section and the MathPiper Programming  
465 Fundamentals section. **A complete list of MathPiper's functions is**  
466 **contained in the MathPiperDocs plugin and more of these functions will**  
467 **be discussed soon.**

## 468 **5.2.2 Accessing Previous Input And Results**

469 The MathPiper console is like a mini text editor which means you can copy text  
470 from it, paste text into it, and edit existing text. You can also reevaluate previous  
471 input by simply placing the cursor on the desired **In>** line and pressing  
472 **<shift><enter>** on it again.

473 The console also keeps a history of all input lines that have been evaluated. If  
474 the cursor is placed on any **In>** line, pressing **<ctrl><up arrow>** will display  
475 each previous line of input that has been entered.

476 Finally, MathPiper associates the most recent computation result with the  
477 percent (%) character. If you want to use the most recent result in a new  
478 calculation, access it with this character:

```
479 In> 5*8  
480 Result> 40
```

```
481 In> %  
482 Result> 40
```

```
483 In> %*2  
484 Result> 80
```

## 485 **5.3 Saving And Restoring A Console Session**

486 If you need to save the contents of a console session, you can copy and paste it  
487 into a MathRider buffer and then save the buffer. You can also copy a console  
488 session out of a previously saved buffer and paste it into the console for further  
489 processing. Section 7 **Using MathRider As A Programmer's Text Editor**  
490 discusses how to use the text editor that is built into MathRider.

### 491 **5.3.1 Syntax Errors**

492 An expression's **syntax** is related to whether it is **typed** correctly or not. If input  
493 is sent to MathPiper which has one or more typing errors in it, MathPiper will  
494 return an error message which is meant to be helpful for locating the error. For  
495 example, if a backwards slash (\) is entered for division instead of a forward slash  
496 (/), MathPiper returns the following error message:

```
497 In> 12 \ 6  
  
498 Error parsing expression, near token \
```

499 The easiest way to fix this problem is to press the **up arrow** key to display the  
500 previously entered line in the console, change the \ to a /, and reevaluate the  
501 expression.

502 This section provided a short introduction to using MathPiper as a numeric  
503 calculator and the next section contains a short introduction to using MathPiper  
504 as a symbolic calculator.

## 505 **5.4 Using The MathPiper Console As A Symbolic Calculator**

506 MathPiper is good at numeric computation, but it is great at symbolic  
507 computation. If you have never used a system that can do symbolic computation,  
508 you are in for a treat!

509 As a first example, lets try adding fractions (which are also called **rational**  
510 **numbers**). Add  $\frac{1}{2} + \frac{1}{3}$  in the MathPiper console:

```
511 In> 1/2 + 1/3  
512 Result> 5/6
```

513 Instead of returning a numeric result like 0.83333333333333333333 (which is  
514 what a scientific calculator would return) MathPiper added these two rational  
515 numbers symbolically and returned  $\frac{5}{6}$  . If you want to work with this result  
516 further, remember that it has also been stored in the % symbol:

```
517 In> %  
518 Result> 5/6
```

519 Lets say that you would like to have MathPiper determine the numerator of this  
520 result. This can be done by using (or **calling**) the **Numerator()** function:

```
521 In> Numerator(%)  
522 Result> 5
```

523 Unfortunately, the % symbol cannot be used to have MathPiper determine the  
524 denominator of  $\frac{5}{6}$  because it only holds the result of the most recent  
525 calculation and  $\frac{5}{6}$  was calculated two steps back.

### 526 **5.4.1 Variables**

527 What would be nice is if MathPiper provided a way to store **results** (which are  
528 also called **values**) in symbols that we choose instead of ones that it chooses.  
529 Fortunately, this is exactly what it does! Symbols that can be associated with  
530 values are called **variables**. Variable names must start with an upper or lower  
531 case letter and be followed by zero or more upper case letters, lower case  
532 letters, or numbers. Examples of variable names include: 'a', 'b', 'x', 'y', 'answer',

533 'totalAmount', and 'loop6'.

534 The process of associating a value with a variable is called **assigning** or **binding**  
535 the value to the variable and this consists of placing the name of a **variable** you  
536 would like to create on the **left** side of an assignment operator (:=) and an  
537 **expression** on the **right** side of this operator. When the expression returns a  
538 value, the value is assigned (or bound to) to the variable.

539 Lets recalculate  $\frac{1}{2} + \frac{1}{3}$  but this time we will assign the result to the variable 'a':

540 In> a := 1/2 + 1/3

541 Result> 5/6

542 In> a

543 Result> 5/6

544 In> Numerator(a)

545 Result> 5

546 In> Denominator(a)

547 Result> 6

548 In this example, the assignment operator (:=) was used to assign the result (or  
549 **value**)  $\frac{5}{6}$  to the variable 'a'. **When 'a' was evaluated by itself, the value it**

550 **was bound to (in this case  $\frac{5}{6}$ ) was returned.** This value will stay bound to

551 the variable 'a' as long as MathPiper is running unless 'a' is cleared with the  
552 **Clear()** function or 'a' has another value assigned to it. This is why we were able  
553 to determine both the numerator and the denominator of the rational number  
554 assigned to 'a' using two functions in turn. **(Note: there can be no spaces**  
555 **between the : and the =)**

#### 556 5.4.1.1 Calculating With Unbound Variables

557 Here is an example which shows another value being assigned to 'a':

558 In> a := 9

559 Result> 9

560 In> a

561 Result> 9

562 and the following example shows 'a' being cleared (or **unbound**) with the  
563 **Clear()** function:

564 In> Clear(a)

```
565 Result> True
```

```
566 In> a
```

```
567 Result> a
```

568 Notice that the `Clear()` function returns '**True**' as a result after it is finished to  
569 indicate that the variable that was sent to it was successfully cleared (or  
570 **unbound**). Many functions either return '**True**' or '**False**' to indicate whether or  
571 not the operation they performed succeeded. Also notice that unbound variables  
572 return themselves when they are evaluated. In this case, 'a' returned 'a'.

573 **Unbound variables** may not appear to be very useful, but they provide the  
574 flexibility needed for computer algebra systems to perform symbolic calculations.  
575 In order to demonstrate this flexibility, let's first factor some numbers using the  
576 **Factor()** function:

```
577 In> Factor(8)
```

```
578 Result> 2^3
```

```
579 In> Factor(14)
```

```
580 Result> 2*7
```

```
581 In> Factor(2343)
```

```
582 Result> 3*11*71
```

583 Now let's factor an expression that contains the unbound variable 'x':

```
584 In> x
```

```
585 Result> x
```

```
586 In> IsBound(x)
```

```
587 Result> False
```

```
588 In> Factor(x^2 + 24*x + 80)
```

```
589 Result> (x+20)*(x+4)
```

```
590 In> Expand(%)
```

```
591 Result> x^2+24*x+80
```

592 Evaluating 'x' by itself shows that it does not have a value bound to it and this  
593 can also be determined by passing 'x' to the **IsBound()** function. `IsBound()`  
594 returns '**True**' if a variable is bound to a value and '**False**' if it is not.

595 What is more interesting, however, are the results returned by **Factor()** and  
596 **Expand()**. **Factor()** is able to determine when expressions with unbound  
597 variables are sent to it and it uses the rules of algebra to **manipulate** them into  
598 factored form. The **Expand()** function was then able to take the factored  
599 expression  $(x+20)(x+4)$  and manipulate it until it was expanded. One way to  
600 remember what the functions **Factor()** and **Expand()** do is to look at the second

601 letters of their names. The 'a' in **Factor** can be thought of as **adding**  
602 parentheses to an expression and the 'x' in **Expand** can be thought of **xing** out  
603 or removing parentheses from an expression.

#### 604 **5.4.1.2 Variable And Function Names Are Case Sensitive**

605 MathPiper variables are **case sensitive**. This means that MathPiper takes into  
606 account the **case** of each letter in a variable name when it is deciding if two or  
607 more variable names are the same variable or not. For example, the variable  
608 name **Box** and the variable name **box** are not the same variable because the first  
609 variable name starts with an upper case 'B' and the second variable name starts  
610 with a lower case 'b':

```
611 In> Box := 1
612 Result> 1
```

```
613 In> box := 2
614 Result> 2
```

```
615 In> Box
616 Result> 1
```

```
617 In> box
618 Result> 2
```

#### 619 **5.4.1.3 Using More Than One Variable**

620 Programs are able to have more than 1 variable and here is a more sophisticated  
621 example which uses 3 variables:

```
622 a := 2
623 Result> 2
```

```
624 b := 3
625 Result> 3
```

```
626 a + b
627 Result> 5
```

```
628 answer := a + b
629 Result> 5
```

```
630 answer
631 Result> 5
```

632 The part of an expression that is on the **right side** of an assignment operator is  
633 always evaluated first and the result is then assigned to the variable that is on

634 the **left side** of the operator.

635 Now that you have seen how to use the MathPiper console as both a **symbolic**  
636 and a **numeric** calculator, our next step is to take a closer look at the functions  
637 which are included with MathPiper. As you will soon discover, MathPiper  
638 contains an amazing number of functions which deal with a wide range of  
639 mathematics.

## 640 **5.5 Exercises**

641 Use the MathPiper console which is at the bottom of the MathRider application  
642 to complete the following exercises.

### 643 **5.5.1 Exercise 1**

644 Carefully read all of section 5. Evaluate each one of the examples in  
645 section 5 in the MathPiper console and verify that the results match the  
646 ones in the book.

### 647 **5.5.2 Exercise 2**

648 Answer each one of the following questions:

649 a) What is the purpose of the N() function?

650 b) What is a variable?

651 c) Are the variables 'x' and 'X' the same variable?

652 d) What is the difference between a bound variable and an unbound variable?

653 e) How can you tell if a variable is bound or not?

654 f) How can a variable be bound to a value?

655 g) How can a variable be unbound from a value?

656 h) What does the % character do?

### 657 **5.5.3 Exercise 3**

658 Perform the following calculation:

$$\frac{1}{4} + \frac{3}{8} - \frac{7}{16}$$

659 **5.5.4 Exercise 4**

660 a) Assign the variable **answer** to the result of the calculation  $\frac{1}{5} + \frac{7}{4} + \frac{15}{16}$

661 using the following line of code:

662 In> **answer** := 1/5 + 7/4 + 15/16

663 b) Use the Numerator() function to calculate the numerator of **answer**.

664 c) Use the Denominator() function to calculate the denominator of **answer**.

665 d) Use the N() function to calculate the numeric value of **answer**.

666 e) Use the Clear() function to unbind the variable **answer** and verify that  
667 **answer** is unbound by executing the following code and by using the  
668 IsBound() function:

669 In> **answer**

670 **5.5.5 Exercise 5**

671 Assign  $\frac{1}{4}$  to variable **x**,  $\frac{3}{8}$  to variable **y**, and  $\frac{7}{16}$  to variable **z** using the  
672 := operator. Then perform the following calculations:

673 a)

674 In> x

675 b)

676 In> y

677 c)

678 In> z

679 d)

680 In> x + y

681 e)

682 In> x + z

683 f)

684 In> x + y + z



## 685 **6 The MathPiper Documentation Plugin**

686 MathPiper has a significant amount of reference documentation written for it  
687 and this documentation has been placed into a plugin called **MathPiperDocs** in  
688 order to make it easier to navigate. The MathPiperDocs plugin is available in a  
689 tab called "MathPiperDocs" which is near the right side of the MathRider  
690 application. Click on this tab to open the plugin and click on it again to close it.

691 The left side of the MathPiperDocs window contains the names of all the  
692 functions that come with MathPiper and the right side of the window contains a  
693 mini-browser that can be used to navigate the documentation.

### 694 **6.1 Function List**

695 MathPiper's functions are divided into two main categories called **user** functions  
696 and **programmer functions**. In general, the **user functions** are used for  
697 solving problems in the MathPiper console or with short programs and the  
698 **programmer functions** are used for longer programs. However, users will  
699 often use some of the programmer functions and programmers will use the user  
700 functions as needed.

701 Both the user and programmer function names have been placed into a "tree" on  
702 the left side of the MathPiperDocs window to allow for easy navigation. The  
703 branches of the function tree can be opened and closed by clicking on the small  
704 "circle with a line attached to it" symbol which is to the left of each branch. Both  
705 the user and programmer branches have the functions they contain organized  
706 into categories and the **top category in each branch** lists all the functions in  
707 the branch in **alphabetical order** for quick access. Clicking on a function will  
708 bring up documentation about it in the browser window and selecting the  
709 **Collapse** button at the top of the plugin will collapse the tree.

710 **Don't be intimidated by the large number of categories and functions**  
711 **that are in the function tree!** Most MathRider beginners will not know what  
712 most of them mean, and some will not know what any of them mean. Part of the  
713 benefit Mathrider provides is exposing the user to the existence of these  
714 categories and functions. The more you use MathRider, the more you will learn  
715 about these categories and functions and someday you may even get to the point  
716 where you understand all of them. This book is designed to show newbies how to  
717 begin using these functions using a gentle step-by-step approach.

### 718 **6.2 Mini Web Browser Interface**

719 MathPiper's reference documentation is in HTML (or web page) format and so  
720 the right side of the plugin contains a mini web browser that can be used to  
721 navigate through these pages. The browser's **home page** contains links to the  
722 main parts of the MathPiper documentation. As links are selected, the **Back** and

723 **Forward** buttons in the upper right corner of the plugin allow the user to move  
724 backward and forward through previously visited pages and the **Home** button  
725 navigates back to the home page.

726 The function names in the function tree all point to sections in the HTML  
727 documentation so the user can access function information either by navigating  
728 to it with the browser or jumping directly to it with the function tree.

## 729 **6.3 Exercises**

### 730 **6.3.1 Exercise 1**

731 Carefully read all of section 6. Locate the `N()`, `IsEven()`, `IsOdd()`,  
732 `Clear()`, `IsBound()`, `Numerator()`, `Denominator()`, and `Factor()` functions in  
733 the **All Functions** section of the MathPiperDocs plugin and read the  
734 information that is available on them. List the one line descriptions  
735 which are at the top of the documentation for each of these functions.

### 736 **6.3.2 Exercise 2**

737 Locate the `N()`, `IsEven()`, `IsOdd()`, `Clear()`, `IsBound()`, `Numerator()`,  
738 `Denominator()`, and `Factor()` functions in the **User Functions** section of the  
739 MathPiperDocs plugin and list which category each function is contained in.  
740 **Don't** include the **Alphabetical** or **Built In** categories in your search. For  
741 example, the `N()` function is in the **Numbers (Operations)** category.

## 742 **7 Using MathRider As A Programmer's Text Editor**

743 We have covered some of MathRider's mathematics capabilities and this section  
744 discusses some of its programming capabilities. As indicated in a previous  
745 section, MathRider is built on top of a programmer's text editor but what wasn't  
746 discussed was what an amazing and powerful tool a programmer's text editor is.

747 Computer programmers are among the most intelligent and productive people in  
748 the world and most of their work is done using a programmer's text editor (or  
749 something similar to one). Programmers have designed programmer's text  
750 editors to be super-tools which can help them maximize their personal  
751 productivity and these tools have all kinds of capabilities that most people would  
752 not even suspect they contained.

753 Even though this book only covers a small part of the editing capabilities that  
754 MathRider has, what is covered will enable the user to begin writing useful  
755 programs.

### 756 **7.1 Creating, Opening, Saving, And Closing Text Files**

757 A good way to begin learning how to use MathRider's text editing capabilities is  
758 by creating, opening, and saving text files. A text file can be created either by  
759 selecting **File->New** from the menu bar or by selecting the icon for this  
760 operation on the tool bar. When a new file is created, an empty text area is  
761 created for it along with a new tab named **Untitled**.

762 The file can be saved by selecting **File->Save** from the menu bar or by selecting  
763 the **Save** icon in the tool bar. The first time a file is saved, MathRider will ask  
764 the user what it should be named and it will also provide a file system navigation  
765 window to determine where it should be placed. After the file has been named  
766 and saved, its name will be shown in the tab that previously displayed **Untitled**.

767 A file can be closed by selecting **File->Close** from the menu bar and it can be  
768 opened by selecting **File->Open**.

### 769 **7.2 Editing Files**

770 If you know how to use a word processor, then it should be fairly easy for you to  
771 learn how to use MathRider as a text editor. Text can be selected by dragging  
772 the mouse pointer across it and it can be cut or copied by using actions in the  
773 **Edit** menu (or by using **<Ctrl>x** and **<Ctrl>c**). Pasting text can be done using  
774 the Edit menu actions or by pressing **<Ctrl>v**.

### 775 **7.3 File Modes**

776 Text file names are suppose to have a file extension which indicates what type of

777 file it is. For example, test.**txt** is a generic text file, test.**bat** is a Windows batch  
778 file, and test.**sh** is a Unix/Linux shell script (**unfortunately, Windows is usually**  
779 **configured to hide file extensions, but viewing a file's properties by right-clicking**  
780 **on it will show this information.**).

781 MathRider uses a file's extension type to set its text area into a customized  
782 **mode** which highlights various parts of its contents. For example, MathRider  
783 worksheet files have a **.mrw** extension and MathRider knows what colors to  
784 highlight the various parts of a .mrw file in.

## 785 ***7.4 Learning How To Type Properly Is An Excellent Investment Of Your*** 786 ***Time***

787 This is a good place in the document to mention that learning how to type  
788 properly is an investment that will pay back dividends throughout your whole  
789 life. Almost any work you do on a computer (including programming) will be  
790 done *much* faster and with less errors if you know how to type properly. Here is  
791 what Steve Yegge has to say about this subject:

792 "If you are a programmer, or an IT professional working with computers in *any*  
793 capacity, **you need to learn to type!** I don't know how to put it any more clearly  
794 than that."

795 A good way to learn how to type is to locate a free "learn how to type" program  
796 on the web and use it.

## 797 ***7.5 Exercises***

### 798 ***7.5.1 Exercise 1***

799 Carefully read all of section 7. Create a text file called  
800 **"my\_text\_file.txt"** and place a few sentences in it. Save the text file  
801 somewhere on your hard drive then close it. Now, open the text file again  
802 using **File->Open** and verify that what you typed is still in the file.

## 803 **8 MathRider Worksheet Files**

804 While MathRider's ability to execute code inside a console provides a significant  
805 amount of power to the user, most of MathRider's power is derived from  
806 **worksheets**. MathRider worksheets are text files which have a **.mrw** extension  
807 and are able to execute multiple types of code in a single text area. The  
808 **worksheet\_demo\_1.mrw** file (which is preloaded in the MathRider environment  
809 when it is first launched) demonstrates how a worksheet is able to execute  
810 multiple types of code in what are called **code folds**.

### 811 **8.1 Code Folds**

812 Code folds are named sections inside a MathRider worksheet which contain  
813 source code that can be executed by placing the cursor inside of it and pressing  
814 **<shift><Enter>**. A fold always begins with a **start tag**, which starts with a  
815 percent symbol (%) followed by the **name of the fold type** (like this:  
816 **%<foldtype>**). The end of a fold is marked by an **end tag** which looks like  
817 **%/<foldtype>**. The only difference between a fold's start tag and its end tag is  
818 that the end tag has a slash (/) after the %.

819 For example, here is a MathPiper fold which will print the result of **2 + 3** to the  
820 MathPiper console (**Note: the semicolon ';' which is at the end of the line of**  
821 **code is required**):

```
822 %mathpiper
823 2 + 3;
824 %/mathpiper
```

825 The **output** generated by a fold (called the **parent fold**) is wrapped in a **new**  
826 **fold** (called a **child fold**) which is indented and placed just below the parent.  
827 This can be seen when the above fold is executed by pressing **<shift><enter>**  
828 inside of it:

```
829 %mathpiper
830 2 + 3;
831 %/mathpiper
832     %output,preserve="false"
833     Result: 5
834 .    %/output
```

835 The most common type of output fold is **%output** and by default folds of type

836 %output have their **preserve property** set to **false**. This tells MathRider to  
837 overwrite the %output fold with a new version during the next execution of its  
838 parent. If preserve is set to **true**, the fold will not be overwritten and a new fold  
839 will be created instead.

840 There are other kinds of child folds, but in the rest of this document they will all  
841 be referred to in general as "output" folds.

### 842 8.1.1 The title Attribute

843 Folds can also have what is called a "**title attribute**" placed after the start tag  
844 which describes what the fold contains. For example, the following %mathpiper  
845 fold has a title attribute which indicates that the fold adds two number together:

```
846 %mathpiper,title="Add two numbers together."
```

```
847 2 + 3;
```

```
848 %/mathpiper
```

849 The title attribute is added to the start tag of a fold by placing a comma after the  
850 fold's type name and then adding the text **title="<text>"** after the comma.  
851 (**Note: no spaces can be present before or after the comma (,) or the**  
852 **equals sign (=)** ).

### 853 8.2 Automatically Inserting Folds & Removing Unpreserved Folds

854 Typing the top and bottom fold lines (for example:

```
855 %mathpiper
```

```
856 %/mathpiper
```

857 can be tedious and MathRider has a way to automatically insert them. Place the  
858 cursor at the beginning of a blank line in a .mrw worksheet file where you would  
859 like a fold inserted and then **press the right mouse button**.

860 A popup menu will be displayed and at the top of this menu are items which read  
861 "**Insert MathPiper Fold**", "**Insert Group Fold**", etc. If you select one of these  
862 menu items, an empty code fold of the proper type will automatically be inserted  
863 into the .mrw file at the position of the cursor.

864 This popup menu also has a menu item called "**Remove Unpreserved Folds**". If  
865 this menu item is selected, all folds which have a "**preserve="false"**" property  
866 will be removed.

### 867 **8.3 Placing Text Outside Of A Fold**

868 Text can also be placed outside of a fold like the following example shows:

869 Text can be placed above folds like this.

```
870 text text text text
```

```
871 text text text text
```

```
872 %mathpiper,title="Fold 1"
```

```
873 2 + 3;
```

```
874 %/mathpiper
```

875 Text can be placed between folds like this.

```
876 text text text text
```

```
877 text text text text
```

```
878 %mathpiper,title="Fold 2"
```

```
879 3 + 4;
```

```
880 %/mathpiper
```

881 Text can be placed between folds like this.

```
882 text text text text
```

```
883 text text text text
```

884 Placing text outside a fold is useful for describing what is being done in certain  
885 folds and it is also good for saving work that has been done in the MathPiper  
886 console.

### 887 **8.4 Exercises**

888 A MathRider worksheet file called "**newbies\_book\_examples\_1.mrw**" can be  
889 obtained from this website:

890 [https://mathrider.dev.java.net/source/browse/mathrider/trunk/src/doc/newbies\\_bo](https://mathrider.dev.java.net/source/browse/mathrider/trunk/src/doc/newbies_book/examples/proposed/misc/newbies_book_examples_1.mrw)  
891 [ok/examples/proposed/misc/newbies\\_book\\_examples\\_1.mrw](https://mathrider.dev.java.net/source/browse/mathrider/trunk/src/doc/newbies_book/examples/proposed/misc/newbies_book_examples_1.mrw)

892 It contains a number of %mathpiper folds which contain code examples from the  
893 previous sections of this book. Notice that all of the lines of code have a  
894 semicolon (;) placed after them. The reason this is needed is explained in a later  
895 section.

896 Download this worksheet file to your computer from the section on this website  
897 that contains the highest revision number and then open it in MathRider. Then,  
898 use the worksheet to do the following exercises.

#### 899 **8.4.1 Exercise 1**

900 Carefully read all of section 8. Execute folds 1-8 in the top section of  
901 the worksheet by placing the cursor inside of the fold and then pressing  
902 <shift><enter> on the keyboard.

#### 903 **8.4.2 Exercise 2**

904 The code in folds 9 and 10 have errors in them. Fix the errors and then  
905 execute the folds again.

#### 906 **8.4.3 Exercise 3**

907 Use the empty fold 11 to calculate the expression  $100 - 23$ ;

#### 908 **8.4.4 Exercise 4**

909 Perform the following calculations by creating new folds at the bottom of  
910 the worksheet (using the right-click popup menu) and placing each  
911 calculation into its own fold:

912 a)  $2 * 7 + 3$

913 b)  $18 / 3$

914 c)  $234238342 + 2038408203$

915 d)  $324802984 * 2308098234$

916 e) Factor the result which was calculated in d).



## 9 MathPiper Programming Fundamentals

The MathPiper language consists of **expressions** and an expression consists of one or more **symbols** which represent **values**, **operators**, **variables**, and **functions**. In this section expressions are explained along with the values, operators, variables, and functions they consist of.

### 9.1 Values and Expressions

A **value** is a single symbol or a group of symbols which represent an idea. For example, the value:

3

represents the number three, the value:

0.5

represents the number one half, and the value:

"Mathematics is powerful!"

represents an English sentence.

Expressions can be created by using **values** and **operators** as building blocks. The following are examples of simple expressions which have been created this way:

3

2 + 3

5 + 6\*21/18 - 2^3

In MathPiper, **expressions** can be **evaluated** which means that they can be transformed into a **result value** by predefined rules. For example, when the expression 2 + 3 is evaluated, the result value that is produced is 5:

In> 2 + 3

Result> 5

### 9.2 Operators

In the above expressions, the characters +, -, \*, /, ^ are called **operators** and their purpose is to tell MathPiper what **operations** to perform on the **values** in an **expression**. For example, in the expression 2 + 3, the **addition** operator + tells MathPiper to add the integer 2 to the integer 3 and return the result.

The **subtraction** operator is -, the **multiplication** operator is \*, / is the **division** operator, % is the **remainder** operator (which is also used as the

949 "result of the last calculation" symbol), and ^ is the **exponent** operator.  
950 MathPiper has more operators in addition to these and some of them will be  
951 covered later.

952 The following examples show the -, \*, /, %, and ^ operators being used:

953 In> 5 - 2  
954 Result> 3

955 In> 3\*4  
956 Result> 12

957 In> 30/3  
958 Result> 10

959 In> 8%5  
960 Result> 3

961 In> 2^3  
962 Result> 8

963 The - character can also be used to indicate a negative number:

964 In> -3  
965 Result> -3

966 Subtracting a negative number results in a positive number (Note: there must be  
967 a space between the two negative signs):

968 In> - -3  
969 Result> 3

970 In MathPiper, **operators** are symbols (or groups of symbols) which are  
971 implemented with **functions**. One can either call the function that an operator  
972 represents directly or use the operator to call the function indirectly. However,  
973 using operators requires less typing and they often make a program easier to  
974 read.

### 975 **9.3 Operator Precedence**

976 When expressions contain more than one operator, MathPiper uses a set of rules  
977 called **operator precedence** to determine the order in which the operators are  
978 applied to the values in the expression. Operator precedence is also referred to  
979 as the **order of operations**. Operators with higher precedence are evaluated  
980 before operators with lower precedence. The following table shows a subset of  
981 MathPiper's operator precedence rules with higher precedence operators being  
982 placed higher in the table:

983       <sup>^</sup>       Exponents are evaluated right to left.

984       \*,%,/ Then multiplication, remainder, and division operations are evaluated  
985       left to right.

986       +, − Finally, addition and subtraction are evaluated left to right.

987   Lets manually apply these precedence rules to the multi-operator expression we  
988   used earlier. Here is the expression in source code form:

989                               5 + 6\*21/18 - 2^3

990   And here it is in traditional form:

$$5 + 6 * \frac{21}{18} - 2^3$$

991   According to the precedence rules, this is the order in which MathPiper  
992   evaluates the operations in this expression:

993   5 + 6\*21/18 - 2^3

994   5 + 6\*21/18 - 8

995   5 + 126/18 - 8

996   5 + 7 - 8

997   12 - 8

998   4

999   Starting with the first expression, MathPiper evaluates the <sup>^</sup> operator first which  
1000   results in the 8 in the expression below it. In the second expression, the \*  
1001   operator is executed next, and so on. The last expression shows that the final  
1002   result after all of the operators have been evaluated is 4.

#### 1003   **9.4 Changing The Order Of Operations In An Expression**

1004   The default order of operations for an expression can be changed by grouping  
1005   various parts of the expression within parentheses (). Parentheses force the  
1006   code that is placed inside of them to be evaluated before any other operators are  
1007   evaluated. For example, the expression 2 + 4\*5 evaluates to 22 using the  
1008   default precedence rules:

1009   In> 2 + 4\*5

1010   Result> 22

1011   If parentheses are placed around 2 + 4, however, the addition operator is forced  
1012   to be evaluated before the multiplication operator and the result is 30:

```
1013 In> (2 + 4)*5
1014 Result> 30
```

1015 Parentheses can also be nested and nested parentheses are evaluated from the  
1016 most deeply nested parentheses outward:

```
1017 In> ((2 + 4)*3)*5
1018 Result> 90
```

1019 (Note: precedence adjusting parentheses are different from the parentheses that  
1020 are used to call functions.)

1021 Since parentheses are evaluated before any other operators, they are placed at  
1022 the top of the precedence table:

- 1023     ()     Parentheses are evaluated from the inside out.
- 1024     ^     Then exponents are evaluated right to left.
- 1025     \*,%,/ Then multiplication, remainder, and division operations are evaluated  
1026           left to right.
- 1027     +, − Finally, addition and subtraction are evaluated left to right.

## 1028 **9.5 Functions & Function Names**

1029 In programming, **functions** are named blocks of code that can be executed one  
1030 or more times by being **called** from other parts of the same program or called  
1031 from other programs. Functions **can have values passed to them** from the  
1032 calling code and they **always return a value** back to the calling code when they  
1033 are finished executing. An example of a function is the **IsEven()** function which  
1034 was discussed in an previous section.

1035 Functions are one way that MathPiper enables code to be reused. Most  
1036 programming languages allow code to be reused in this way, although in other  
1037 languages these named blocks of code are sometimes called **subroutines**,  
1038 **procedures**, or **methods**.

1039 The functions that come with MathPiper have names which consist of either a  
1040 single word (such as **Sum()**) or multiple words that have been put together to  
1041 form a compound word (such as **IsBound()**). All letters in the names of  
1042 functions which come with MathPiper are lower case except the beginning letter  
1043 in each word, which are upper case.

## 1044 **9.6 Functions That Produce Side Effects**

1045 Most functions are executed to obtain the **results** they produce but some  
1046 functions are executed in order to **have them perform work that is not in the**  
1047 **form of a result**. Functions that perform work that is not in the form of a result  
1048 are said to produce **side effects**. Side effects include many forms of work such  
1049 as sending information to the user, opening files, and changing values in the  
1050 computer's memory.

1051 When a function produces a side effect which sends information to the user, this  
1052 information has the words **Side Effects:** placed before it in the output instead of  
1053 the word **Result:**. The **Echo()** and **Write()** functions are examples of functions  
1054 that produce side effects and they are covered in the next section.

### 1055 **9.6.1 Printing Related Functions: Echo(), Write(), And Newline()**

1056 The printing related functions send text information to the user and this is  
1057 usually referred to as "printing" in this document. However, it may also be called  
1058 "echoing" and "writing".

#### 1059 **9.6.1.1 Echo()**

1060 The **Echo()** function takes one expression (or multiple expressions separated by  
1061 commas) evaluates each expression, and then prints the results as side effect  
1062 output. The following examples illustrate this:

```
1063 In> Echo(1)
1064 Result> True
1065 Side Effects>
1066 1
```

1067 In this example, the number 1 was passed to the Echo() function, the number  
1068 was evaluated (all numbers evaluate to themselves), and the result of the  
1069 evaluation was then printed as a side effect. Notice that Echo() **also returned a**  
1070 **result**. In MathPiper, all functions return a result, but functions whose main  
1071 purpose is to produce a side effect usually just return a result of **True** if the side  
1072 effect succeeded or **False** if it failed. In this case, Echo() returned a result of  
1073 **True** because it was able to successfully print a 1 as its side effect.

1074 The next example shows multiple expressions being sent to Echo() (notice that  
1075 the expressions are separated by commas):

```
1076 In> Echo(1,1+2,2*3)
1077 Result> True
1078 Side Effects>
1079 1 3 6
```

1080 The expressions were each evaluated and their results were returned (separated  
1081 by spaces) as side effect output. If it is desired that commas be printed between  
1082 the numbers in the output, simply place three commas between the expressions  
1083 that are passed to Echo():

```
1084 In> Echo(1,,,1+2,,,2*3)
1085 Result> True
1086 Side Effects>
1087 1 , 3 , 6
```

1088 Each time an Echo() function is executed, it always forces the display to drop  
1089 down to the next line after it is finished. This can be seen in the following  
1090 program which is similar to the previous one except it uses a separate Echo()  
1091 function to display each expression:

```
1092 %mathpiper
1093 Echo(1);
1094 Echo(1+2);
1095 Echo(2*3);
1096 %/mathpiper
1097 %output,preserve="false"
1098 Result: True
1099
1100 Side Effects:
1101 1
1102 3
1103 6
1104 . %/output
```

1105 Notice how the 1, the 3, and the 6 are each on their own line.

1106 Now that we have seen how Echo() works, lets use it to do something useful. If  
1107 more than one expression is evaluated in a %mathpiper fold, only the result from  
1108 the last expression that was evaluated (which is usually the bottommost  
1109 expression) is displayed:

```
1110 %mathpiper
1111 a := 1;
1112 b := 2;
1113 c := 3;
1114 %/mathpiper
```

```
1115     %output,preserve="false"
1116     Result: 3
1117 .    %/output
```

1118 In MathPiper, programs are executed one line at a time, starting at the topmost  
1119 line of code and working downwards from there. In this example, the line `a := 1;`  
1120 is executed first, then the line `b := 2;` is executed, and so on. Notice, however,  
1121 that even though we wanted to see what was in all three variables, only the  
1122 content of the last variable was displayed.

1123 The following example shows how `Echo()` can be used to display the contents of  
1124 all three variables:

```
1125 %mathpiper
1126 a := 1;
1127 Echo(a);
1128 b := 2;
1129 Echo(b);
1130 c := 3;
1131 Echo(c);
1132 %/mathpiper
1133     %output,preserve="false"
1134     Result: True
1135
1136     Side Effects:
1137     1
1138     2
1139     3
1140 .    %/output
```

#### 1141 9.6.1.2 Echo Functions Are Useful For "Debugging" Programs

1142 The errors that are in a program are often called "bugs". This name came from  
1143 the days when computers were the size of large rooms and were made using  
1144 electromechanical parts. Periodically, bugs would crawl into the machines and  
1145 interfere with its moving mechanical parts and this would cause the machine to  
1146 malfunction. The bugs needed to be located and removed before the machine  
1147 would run properly again.

1148 Of course, even back then most program errors were produced by programmers  
1149 entering wrong programs or entering programs wrong, but they liked to say that  
1150 all of the errors were caused by bugs and not by themselves! The process of  
1151 fixing errors in a program became known as **debugging** and the names "bugs"

1152 and "debugging" are still used by programmers today.

1153 One of the standard ways to locate bugs in a program is to place **Echo()** function  
1154 calls in the code at strategic places which **print the contents of variables and**  
1155 **display messages**. These Echo() functions will enable you to see what your  
1156 program is doing while it is running. After you have found and fixed the bugs in  
1157 your program, you can remove the debugging Echo() function calls or comment  
1158 them out if you think they may be needed later.

### 1159 9.6.1.3 Write()

1160 The **Write()** function is similar to the Echo() function except it does not  
1161 automatically drop the display down to the next line after it finishes executing:

```
1162 %mathpiper
1163 Write(1,,);
1164 Write(1+2,,);
1165 Echo(2*3);
1166 %/mathpiper
1167     %output,preserve="false"
1168     Result: True
1169
1170     Side Effects:
1171     1,3,6
1172 .    %/output
```

1173 Write() and Echo() have other differences besides the one discussed here and  
1174 more information about them can be found in the documentation for these  
1175 functions.

### 1176 9.6.1.4 NewLine()

1177 The **NewLine()** function simply prints a blank line in the side effects output. It  
1178 is useful for placing vertical space between printed lines:

```
1179 %mathpiper
1180 a := 1;
1181 Echo(a);
1182 NewLine();
1183 b := 2;
1184 Echo(b);
```



```
1185 NewLine();
1186 c := 3;
1187 Echo(c);

1188 %mathpiper

1189     %output,preserve="false"
1190     Result: True
1191
1192     Side Effects:
1193     1
1194
1195     2
1196
1197     3
1198 .    %/output
```

## 1197 9.7 Expressions Are Separated By Semicolons

1198 As discussed earlier, all of the expressions that are inside of a **%mathpiper** fold  
1199 must have a semicolon (;) after them. However, the expressions executed in the  
1200 **MathPiper console** did not have a semicolon after them. MathPiper actually  
1201 requires that all expressions end with a semicolon, but one does not need to add  
1202 a semicolon to an expression which is typed into the MathPiper console **because**  
1203 **the console adds it automatically** when the expression is executed.

### 1204 9.7.1 Placing More Than One Expression On A Line In A Fold

1205 All the previous code examples have had each of their expressions on a separate  
1206 line, but multiple expressions can also be placed on a single line because the  
1207 semicolons tell MathPiper where one expression ends and the next one begins:

```
1208 %mathpiper

1209 a := 1; Echo(a); b := 2; Echo(b); c := 3; Echo(c);

1210 %/mathpiper

1211     %output,preserve="false"
1212     Result: True
1213
1214     Side Effects:
1215     1
1216     2
1217     3
1218 .    %/output
```

1219 The spaces that are in the code of this example are used to make the code more  
1220 readable. Any spaces that are present within any expressions or between them  
1221 are ignored by MathPiper and if we remove the spaces from the previous code,  
1222 the output remains the same:

```
1223 %mathpiper
1224 a:=1;Echo(a);b:=2;Echo(b);c:= 3;Echo(c);
1225 %/mathpiper
1226     %output,preserve="false"
1227     Result: True
1228
1229     Side Effects:
1230     1
1231     2
1232     3
1233 .    %/output
```

## 1234 9.7.2 Placing Multiple Expressions In A Code Block

1235 A **code block** (which is also called a **compound expression**) consists of one or  
1236 more expressions which are separated by semicolons and placed within an open  
1237 bracket (**[**) and close bracket (**]**) pair. When a code block is evaluated, each  
1238 expression in the block will be executed from left to right. The following  
1239 example shows expressions being executed within of a code block inside the  
1240 MathPiper console:

```
1241 In> [a := 1; Echo(a); b := 2; Echo(b); c := 3; Echo(c);]
1242 Result> True
1243 Side Effects>
1244 1
1245 2
1246 3
```

1247 Notice that all of the expressions were executed and 1-3 was printed as a side  
1248 effect. Code blocks **always return the result of the last expression executed**  
1249 **as the result of the whole block**. In this case, **True** was returned as the result  
1250 because the last **Echo(c)** function returned **True**. If we place **another**  
1251 **expression after the Echo(c) function**, however, **the block will execute this**  
1252 **new expression last and its result will be the one returned by the block**:

```
1253 In> [a := 1; Echo(a); b := 2; Echo(b); c := 3; Echo(c); 2 + 2;]
1254 Result> 4
1255 Side Effects>
1256 1
```

1257 2  
1258 3

1259 Finally, code blocks can have their contents placed on separate lines if desired:

```
1260 %mathpiper
1261 [
1262     a := 1;
1263
1264     Echo(a);
1265
1266     b := 2;
1267
1268     Echo(b);
1269
1270     c := 3;
1271
1272     Echo(c);
1273 ];
1274
1275 %output,preserve="false"
1276 Result: True
1277
1278 Side Effects:
1279 1
1280 2
1281 3
1282 . %/output
```

1283 Code blocks are very powerful and we will be discussing them further in later  
1284 sections.

### 1285 9.7.2.1 Automatic Bracket, Parentheses, And Brace Match Indicating

1286 In programming, most open brackets '[' have a close bracket ']', most open  
1287 parentheses '(' have a close parentheses ')', and most open braces '{' have a  
1288 close brace '}'. It is often difficult to make sure that each "open" character has a  
1289 matching "close" character and if any of these characters don't have a match,  
1290 then an error will be produced.

1291 Thankfully, most programming text editors have a character match indicating  
1292 tool that will help locate problems. To try this tool, paste the following code into  
1293 a .mrw file and following the directions that are present in its comments:

```
1294 %mathpiper
1295 /*
```

```
1296      Copy this code into a .mrw file. Then, place the cursor
1297      to the immediate right of any {, }, [, ], (, or ) character.
1298      You should notice that the match to this character is
1299      indicated by a rectangle being drawing around it.
1300      */
```

```
1301  list := {1,2,3};
1302  [
1303      Echo("Hello");
1304      Echo(list);
1305  ];
1306  %/mathpiper
```

## 1307 9.8 Strings

1308 A **string** is a **value** that is used to hold text-based information. The typical  
1309 expression that is used to create a string consists of **text which is enclosed**  
1310 **within double quotes**. Strings can be assigned to variables just like numbers  
1311 can and strings can also be displayed using the Echo() function. The following  
1312 program assigns a string value to the variable 'a' and then echos it to the user:

```
1313 %mathpiper
1314 a := "Hello, I am a string.";
1315 Echo(a);
1316 %/mathpiper
1317 %output,preserve="false"
1318 Result: True
1319
1320 Side Effects:
1321 Hello, I am a string.
1322 . %/output
```

### 1323 9.8.1 The MathPiper Console and MathPiper Folds Can Access The Same 1324 Variables

1325 A useful aspect of using MathPiper inside of MathRider is that variables that are  
1326 assigned inside of a **%mathpiper fold** are accessible inside of the **MathPiper**  
1327 **console** and variables that are assigned inside of the **MathPiper console** are  
1328 available inside of **%mathpiper folds**. For example, after the above fold is  
1329 executed, the string that has been bound to variable 'a' can be displayed in the  
1330 MathPiper console:

```
1331 In> a
1332 Result> "Hello, I am a string."
```

## 1333 9.8.2 Using Strings To Make Echo's Output Easier To Read

1334 When the Echo() function is used to print the values of multiple variables, it is  
1335 often helpful to print some information next to each variable so that it is easier to  
1336 determine which value came from which Echo() function in the code. The  
1337 following program prints the name of the variable that each value came from  
1338 next to it in the side effects output:

```
1339 %mathpiper
1340 a := 1;
1341 Echo("Variable a: ", a);
1342 b := 2;
1343 Echo("Variable b: ", b);
1344 c := 3;
1345 Echo("Variable c: ", c);
1346 %/mathpiper
1347 %output,preserve="false"
1348 Result: True
1349
1350 Side Effects:
1351 Variable a: 1
1352 Variable b: 2
1353 Variable c: 3
1354 . %/output
```

### 1355 9.8.2.1 Combining Strings With The : Operator

1356 If you need to combine two or more strings into one string, you can use the :  
1357 operator like this:

```
1358 In> "A" : "B" : "C"
1359 Result: "ABC"
1360 In> "Hello " : "there!"
1361 Result: "Hello there!"
```

### 1362 9.8.2.2 WriteString()

1363 The **WriteString()** function prints a string without showing the double quotes

1364 that are around it.. For example, here is the Write() function being used to print  
1365 the string "Hello":

```
1366 In> Write("Hello")
1367 Result: True
1368 Side Effects:
1369 "Hello"
```

1370 Notice the double quotes? Here is how the WriteString() function prints "Hello":

```
1371 In> WriteString("Hello")
1372 Result: True
1373 Side Effects:
1374 Hello
```

### 1375 **9.8.2.3 NI()**

1376 The **NI()** (New Line) function is used with the : function to place newline  
1377 characters inside of strings:

```
1378 In> WriteString("A": NI() : "B")
1379 Result: True
1380 Side Effects:
1381 A
1382 B
```

### 1383 **9.8.2.4 Space()**

1384 The Space() function is used to add spaces to printed output:

```
1385 In> WriteString("A"); Space(5); WriteString("B")
1386 Result: True
1387 Side Effects:
1388 A      B
```

```
1389 In> WriteString("A"); Space(10); WriteString("B")
1390 Result: True
1391 Side Effects:
1392 A          B
```

```
1393 In> WriteString("A"); Space(20); WriteString("B")
1394 Result: True
1395 Side Effects:
1396 A                      B
```

## 1397 **9.8.3 Accessing The Individual Letters In A String**

1398 Individual letters in a string (which are also called **characters**) can be accessed  
1399 by placing the character's position number (also called an **index**) inside of

1400 brackets **[]** after the variable it is bound to. A character's position is determined  
1401 by its distance from the left side of the string starting at 1. For example, in the  
1402 string "Hello", 'H' is at position 1, 'e' is at position 2, etc. The following code  
1403 shows individual characters in the above string being accessed:

```
1404 In> a := "Hello, I am a string."  
1405 Result> "Hello, I am a string."
```

```
1406 In> a[1]  
1407 Result> "H"
```

```
1408 In> a[2]  
1409 Result> "e"
```

```
1410 In> a[3]  
1411 Result> "l"
```

```
1412 In> a[4]  
1413 Result> "l"
```

```
1414 In> a[5]  
1415 Result> "o"
```

#### 1416 **9.8.3.1 Indexing Before The Beginning Of A String Or Past The End Of A String**

1417 Lets see what happens if an index is used that is less than **1** or greater than the  
1418 length of a given string. First, we will bind the string "Hello" to the variable 'a':

```
1419 In> a := "Hello"  
1420 Result: "Hello"
```

1421 Then, we'll index the character at position **1** and then the character at position **0**:

```
1422 In> a[1]  
1423 Result: "H"
```

```
1424 In> a[0]  
1425 Result:  
1426 Exception: In function "StringMidGet" :  
1427 bad argument number 1(counting from 1) :  
  
1428 The offending argument aindex evaluated to 0
```

1429 Notice that using an index of **0** resulted in an error.

1430 Next, lets access the character at position **5** (which is the 'o'), then the character  
1431 at position **6** and finally the character at position **7**:

```
1432 In> a[5]
```

1433 Result: "o"

1434 In> a[6]

1435 Result: ""

1436 In> a[7]

1437 Result:

1438 Exception: String index out of range: 8

1439 The 'o' at position **5** was returned correctly, but accessing position **6** returned a  
1440 double quote character (") and accessing position 7 resulted in an error. What  
1441 you can see in this section is that errors are usually produced if an index is not  
1442 set to the position of an actual character in a string.

## 1443 9.9 Comments

1444 Source code can often be difficult to understand and therefore all programming  
1445 languages provide the ability for **comments** to be included in the code.

1446 Comments are used to explain what the code near them is doing and they are  
1447 usually meant to be read by humans instead of being processed by a computer.  
1448 Therefore, comments are ignored by the computer when a program is executed.

1449 There are two ways that MathPiper allows comments to be added to source code.  
1450 The first way is by placing two forward slashes // to the left of any text that is  
1451 meant to serve as a comment. The text from the slashes to the end of the line  
1452 the slashes are on will be treated as a comment. Here is a program that contains  
1453 comments which use slashes:

1454 %mathpiper

1455 //This is a comment.

1456 x := 2; //Set the variable x equal to 2.

1457 %/mathpiper

1458 %output,preserve="false"

1459 Result: 2

1460 . %/output

1461 When this program is executed, any text that starts with slashes is ignored.

1462 The second way to add comments to a MathPiper program is by enclosing the  
1463 comments inside of slash-asterisk/asterisk-slash symbols /\* \*/. This option is  
1464 useful when a comment is too large to fit on one line. Any text between these  
1465 symbols is ignored by the computer. This program shows a longer comment  
1466 which has been placed between these symbols:



```
1467 %mathpiper
1468 /*
1469  This is a longer comment and it uses
1470  more than one line. The following
1471  code assigns the number 3 to variable
1472  x and then returns it as a result.
1473 */
1474 x := 3;
1475 %/mathpiper
1476     %output,preserve="false"
1477     Result: 3
1478 .    %/output
```

## 1479 **9.10 How To Tell If MathPiper Has Crashed And What To Do If It Has**

1480 Sometimes code will be evaluated which has one or more unusual errors in it and  
1481 the errors will cause MathPiper to "crash". Unfortunately, beginners are more  
1482 likely to crash MathPiper than more experienced programmers are because a  
1483 beginner's program is more likely to have errors in it. When MathPiper crashes,  
1484 no harm is done but it will not work correctly after that. **The only way to  
1485 recover from a MathPiper crash is to exit MathRider and then relaunch  
1486 it.** All the information in your buffers will be saved and preserved **but the  
1487 contents of the console will not be.** Be sure to copy the contents of the  
1488 console into a buffer and then save it before restarting.

1489 The main way to tell if MathRider has crashed is that it will indicate that **there  
1490 are errors in lines of code that are actually fine.** If you are receiving an  
1491 error in code that looks okay to you, simply restarting MathRider may fix the  
1492 problem. If you restart MathRider and the error is still present, this usually  
1493 means that there really is an error in the code.

## 1494 **9.11 Exercises**

1495 For the following exercises, create a new MathRider worksheet file called  
1496 **book\_1\_section\_9\_exercises\_<your first name>\_<your last name>.mrw.**  
1497 **(Note: there are no spaces in this file name).** For example, John Smith's  
1498 worksheet would be called:

1499 **book\_1\_section\_9\_exercises\_john\_smith.mrw.**

1500 After this worksheet has been created, place your answer for each exercise that  
1501 requires a fold into its own fold in this worksheet. Place a title attribute in the  
1502 start tag of each fold which indicates the exercise the fold contains the solution  
1503 to. The folds you create should look similar to this one:

1504 `%mathpiper,title="Exercise 1"`

1505 `//Sample fold.`

1506 `%/mathpiper`

1507 If an exercise uses the MathPiper console instead of a fold, copy the work you  
1508 did in the console into the worksheet so it can be saved.

### 1509 **9.11.1 Exercise 1**

1510 Carefully read all of section 9. Evaluate each one of the examples in  
1511 section 9 in the MathPiper worksheet you created or in the MathPiper  
1512 console and verify that the results match the ones in the book. Copy all  
1513 of the console examples you evaluated into your worksheet so they will be  
1514 saved but do not put them in a fold.

### 1515 **9.11.2 Exercise 2**

1516 Change the precedence of the following expression using parentheses so that  
1517 it prints 20 instead of 14:

1518 `2 + 3 * 4`

### 1519 **9.11.3 Exercise 3**

1520 Place the following calculations into a fold and then use one Echo()  
1521 function per variable to print the results of the calculations. Put  
1522 strings in the Echo() functions which indicate which variable each  
1523 calculated value is bound to:

1524 `a := 1+2+3+4+5;`

1525 `b := 1-2-3-4-5;`

1526 `c := 1*2*3*4*5;`

1527 `d := 1/2/3/4/5;`

### 1528 **9.11.4 Exercise 4**

1529 Place the following calculations into a fold and then use one Echo()  
1530 function to print the results of all the calculations on a single line  
1531 (Remember, the Echo() function can print multiple values if they are  
1532 separated by commas.):

1533 `Clear(x);`

1534 `a := 2*2*2*2*2;`

1535 `b := 2^5;`

1536 `c := x^2 * x^3;`

1537 `d := 2^2 * 2^3;`

1538 **9.11.5 Exercise 5**

1539 The following code assigns a string which contains all of the upper case  
1540 letters of the alphabet to the variable **upper**. Each of the three Echo()  
1541 functions prints an index number and the letter that is at that position in  
1542 the string. Place this code into a fold and then continue the Echo()  
1543 functions so that all 26 letters and their index numbers are printed

```
1544 upper := "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

```
1545 Echo(1,upper[1]);
```

```
1546 Echo(2,upper[2]);
```

```
1547 Echo(3,upper[3]);
```

1548 **9.11.6 Exercise 6**

1549 Use Echo() functions to print an index number and the character at this  
1550 position for the following string (this is similar to what was done in the  
1551 previous exercise.):

```
1552 extra := "!.@#$$%^&*() _+<>,?/{ }[]|\-=;";
```

```
1553 Echo(1,extra[1]);
```

```
1554 Echo(2,extra[2]);
```

```
1555 Echo(3,extra[3]);
```

1556 **9.11.7 Exercise 7**

1557 The following program uses strings and index numbers to print a person's  
1558 name. Create a program which uses the three strings from this program to  
1559 print the names of three of your favorite musical bands.

```
1560 %mathpiper
```

```
1561 /*
```

```
1562     This program uses strings and index numbers to print
```

```
1563     a person's name.
```

```
1564 */
```

```
1565 upper := "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

```
1566 lower := "abcdefghijklmnopqrstuvwxyz";
```

```
1567 extra := "!.@#$$%^&*() _+<>,?/{ }[]|\-=;";
```

```
1568 //Print "Mary Smith."
```

```
1569 Echo(upper[13],lower[1],lower[18],lower[25],extra[12],upper[19],lower[13],l
```

```
1570 ower[9],lower[20],lower[8],extra[1]);
```

```
1571 %/mathpiper
```

```
1572     %output,preserve="false"
```

```
1573         Result: True
1574
1575         Side Effects:
1576         Mary Smith.
1577     .    %/output
```

## 1578 10 Rectangular Selection Mode And Text Area Splitting

### 1579 10.1 Rectangular Selection Mode

1580 One capability that MathRider has that a word processor may not have is the  
1581 ability to select rectangular sections of text. To see how this works, do the  
1582 following:

- 1583 1) Type three or four lines of text into a text area.
- 1584 2) Hold down the **<Alt>** key then slowly press the **backslash key** (\) a few  
1585 times. The bottom of the MathRider window contains a text field which  
1586 MathRider uses to communicate information to the user. As **<Alt>\** is  
1587 repeatedly pressed, messages are displayed which read **Rectangular**  
1588 **selection is on** and **Rectangular selection is off**.
- 1589 3) Turn rectangular selection on and then select some text in order to see  
1590 how this is different than normal selection mode. **When you are done**  
1591 **experimenting, set rectangular selection mode to off.**

1592 Most of the time normal selection mode is what you want to use but in certain  
1593 situations rectangular selection mode is better.

### 1594 10.2 Text area splitting

1595 Sometimes it is useful to have two or more text areas open for a single document  
1596 or multiple documents so that different parts of the documents can be edited at  
1597 the same time. A situation where this would have been helpful was in the  
1598 previous section where the output from an exercise in a MathRider worksheet  
1599 contained a list of index numbers and letters which was useful for completing a  
1600 later exercise.

1601 MathRider has this ability and it is called **splitting**. If you look just to the right  
1602 of the toolbar there is an icon which looks like a blank window, an icon to the  
1603 right of it which looks like a window which was split horizontally, and an icon to  
1604 the right of the horizontal one which is split vertically. If you let your mouse  
1605 hover over these icons, a short description will be displayed for each of them.

1606 Select a text area and then experiment with splitting it by pressing the horizontal  
1607 and vertical splitting buttons. Move around these split text areas with their  
1608 scroll bars and when you want to unsplit the document, just press the "**Unsplit**  
1609 **All**" icon.

### 1610 10.3 Exercises

1611 For the following exercises, create a new MathRider worksheet file called  
1612 **book\_1\_section\_10\_exercises\_<your first name>\_<your last name>.mrw**.

1613 (**Note: there are no spaces in this file name**). For example, John Smith's  
1614 worksheet would be called:

1615 **book\_1\_section\_10\_exercises\_john\_smith.mrw.**

1616 For the following exercises, simply type your answers anywhere in the  
1617 worksheet.

### 1618 **10.3.1 Exercise 1**

1619 Carefully read all of section 10 then answer the following questions:

1620 a) Give two examples where rectangular selection mode may be more useful  
1621 than regular selection mode.

1622 b) How can windows that have been split be unsplit?

## 11 Working With Random Integers

It is often useful to use random integers in a program. For example, a program may need to simulate the rolling of dice in a game. In this section, a function for obtaining nonnegative integers is discussed along with how to use it to simulate the rolling of dice.

### 11.1 Obtaining Random Integers With The RandomInteger() Function

One way that a MathPiper program can generate random integers is with the **RandomInteger()** function. The **RandomInteger()** function takes an integer as a parameter and it returns a random integer between 1 and the passed in integer. The following example shows random integers between 1 and 5 **inclusive** being obtained from **RandomInteger()**. **Inclusive** here means that both 1 and 5 are included in the range of random integers that may be returned. If the word **exclusive** was used instead, this would mean that neither 1 nor 5 would be in the range.

```
In> RandomInteger(5)
Result> 4
In> RandomInteger(5)
Result> 5
In> RandomInteger(5)
Result> 4
In> RandomInteger(5)
Result> 2
In> RandomInteger(5)
Result> 3
In> RandomInteger(5)
Result> 5
In> RandomInteger(5)
Result> 2
In> RandomInteger(5)
Result> 2
In> RandomInteger(5)
Result> 1
In> RandomInteger(5)
Result> 2
```

Random integers between 1 and 100 can be generated by passing 100 to **RandomInteger()**:

```
In> RandomInteger(100)
Result> 15
In> RandomInteger(100)
Result> 14
```

```
1663 In> RandomInteger(100)
1664 Result> 82
1665 In> RandomInteger(100)
1666 Result> 93
1667 In> RandomInteger(100)
1668 Result> 32
```

1669 A range of random integers that does not start with 1 can also be generated by  
1670 using the **two argument** version of **RandomInteger()**. For example, random  
1671 integers between 25 and 75 can be obtained by passing RandomInteger() the  
1672 lowest integer in the range and the highest one:

```
1673 In> RandomInteger(25, 75)
1674 Result: 28
1675 In> RandomInteger(25, 75)
1676 Result: 37
1677 In> RandomInteger(25, 75)
1678 Result: 58
1679 In> RandomInteger(25, 75)
1680 Result: 50
1681 In> RandomInteger(25, 75)
1682 Result: 70
```

## 1683 **11.2 Simulating The Rolling Of Dice**

1684 The following example shows the simulated rolling of a single six sided die using  
1685 the RandomInteger() function:

```
1686 In> RandomInteger(6)
1687 Result> 5
1688 In> RandomInteger(6)
1689 Result> 6
1690 In> RandomInteger(6)
1691 Result> 3
1692 In> RandomInteger(6)
1693 Result> 2
1694 In> RandomInteger(6)
1695 Result> 5
```

1696 Code that simulates the rolling of two 6 sided dice can be evaluated in the  
1697 MathPiper console by placing it within a **code block**. The following code  
1698 outputs the sum of the two simulated dice:

```
1699 In> [a := RandomInteger(6); b := RandomInteger(6); a + b;]
1700 Result> 6
1701 In> [a := RandomInteger(6); b := RandomInteger(6); a + b;]
1702 Result> 12
1703 In> [a := RandomInteger(6); b := RandomInteger(6); a + b;]
1704 Result> 6
```



```
1705 In> [a := RandomInteger(6); b := RandomInteger(6); a + b;]
1706 Result> 4
1707 In> [a := RandomInteger(6); b := RandomInteger(6); a + b;]
1708 Result> 3
1709 In> [a := RandomInteger(6); b := RandomInteger(6); a + b;]
1710 Result> 8
```

1711 Now that we have the ability to simulate the rolling of two 6 sided dice, it would  
1712 be interesting to determine if some sums of these dice occur more frequently  
1713 than other sums. What we would like to do is to roll these simulated dice  
1714 hundreds (or even thousands) of times and then analyze the sums that were  
1715 produced. We don't have the programming capability to easily do this yet, but  
1716 after we finish the section on **while loops**, we will.

### 1717 11.3 Exercises

1718 For the following exercises, create a new MathRider worksheet file called  
1719 **book\_1\_section\_11\_exercises\_<your first name>\_<your last name>.mrw**.  
1720 (**Note: there are no spaces in this file name**). For example, John Smith's  
1721 worksheet would be called:

1722 **book\_1\_section\_11\_exercises\_john\_smith.mrw**.

1723 After this worksheet has been created, place your answer for each exercise that  
1724 requires a fold into its own fold in this worksheet. Place a title attribute in the  
1725 start tag of each fold which indicates the exercise the fold contains the solution  
1726 to. The folds you create should look similar to this one:

```
1727 %mathpiper,title="Exercise 1"
1728 //Sample fold.
1729 %/mathpiper
```

1730 If an exercise uses the MathPiper console instead of a fold, copy the work you  
1731 did in the console into the worksheet so it can be saved but do not put it in a fold.

#### 1732 11.3.1 Exercise 1

1733 Carefully read all of section 11. Evaluate each one of the examples in  
1734 section 11 in the MathPiper worksheet you created or in the MathPiper  
1735 console and verify that the results match the ones in the book. Copy all  
1736 of the console examples you evaluated into your worksheet so they will be  
1737 saved but do not put them in a fold.

## 12 Making Decisions

The simple programs that have been discussed up to this point show some of the power that software makes available to programmers. However, these programs are limited in their problem solving ability because they are unable to make decisions. This section shows how programs which have the ability to make decisions are able to solve a wider range of problems than programs that can't make decisions.

### 12.1 Conditional Operators

A program's decision making ability is based on a set of special operators which are called **conditional operators**. A **conditional operator** is an operator that is used to **compare two values**. Expressions that contain conditional operators return a **boolean value** and a **boolean value** is one that can only be **True** or **False**. In case you are curious about the strange name, boolean values come from the area of mathematics called **boolean logic**. This logic was created by a mathematician named **George Boole** and this is where the name boolean came from. Table 2 shows the conditional operators that MathPiper uses:

Operator	Description
<code>x = y</code>	Returns <b>True</b> if the two values are equal and <b>False</b> if they are not equal. Notice that <code>=</code> performs a comparison and not an assignment like <code>:=</code> does.
<code>x != y</code>	Returns <b>True</b> if the values are not equal and <b>False</b> if they are equal.
<code>x &lt; y</code>	Returns <b>True</b> if the left value is less than the right value and <b>False</b> if the left value is not less than the right value.
<code>x &lt;= y</code>	Returns <b>True</b> if the left value is less than or equal to the right value and <b>False</b> if the left value is not less than or equal to the right value.
<code>x &gt; y</code>	Returns <b>True</b> if the left value is greater than the right value and <b>False</b> if the left value is not greater than the right value.
<code>x &gt;= y</code>	Returns <b>True</b> if the left value is greater than or equal to the right value and <b>False</b> if the left value is not greater than or equal to the right value.

Table 2: Conditional Operators

This example shows some of these conditional operators being evaluated in the MathPiper console:

```
In> 1 < 2
Result> True
```

```
1758 In> 4 > 5
1759 Result> False
```

```
1760 In> 8 >= 8
1761 Result> True
```

```
1762 In> 5 <= 10
1763 Result> True
```

1764 The following examples show each of the conditional operators in Table 2 being  
1765 used to compare values that have been assigned to variables **x** and **y**:

```
1766 %mathpiper
```

```
1767 // Example 1.
1768 x := 2;
1769 y := 3;
```

```
1770 Echo(x, "=", y, ":", x = y);
1771 Echo(x, "!= ", y, ":", x != y);
1772 Echo(x, "< ", y, ":", x < y);
1773 Echo(x, "<= ", y, ":", x <= y);
1774 Echo(x, "> ", y, ":", x > y);
1775 Echo(x, ">= ", y, ":", x >= y);
```

```
1776 %/mathpiper
```

```
1777 %output,preserve="false"
1778 Result: True
1779
1780 Side Effects:
1781 2 = 3 :False
1782 2 != 3 :True
1783 2 < 3 :True
1784 2 <= 3 :True
1785 2 > 3 :False
1786 2 >= 3 :False
1787 . %/output
```

```
1788 %mathpiper
```

```
1789 // Example 2.
1790 x := 2;
1791 y := 2;
```

```
1792 Echo(x, "=", y, ":", x = y);
1793 Echo(x, "!= ", y, ":", x != y);
1794 Echo(x, "< ", y, ":", x < y);
1795 Echo(x, "<= ", y, ":", x <= y);
1796 Echo(x, "> ", y, ":", x > y);
```

```
1797     Echo(x, ">= ", y, ":", x >= y);
```

```
1798 %/mathpiper
```

```
1799     %output,preserve="false"
```

```
1800     Result: True
```

```
1801
```

```
1802     Side Effects:
```

```
1803     2 = 2 :True
```

```
1804     2 != 2 :False
```

```
1805     2 < 2 :False
```

```
1806     2 <= 2 :True
```

```
1807     2 > 2 :False
```

```
1808     2 >= 2 :True
```

```
1809 .    %/output
```

```
1810 %mathpiper
```

```
1811 // Example 3.
```

```
1812 x := 3;
```

```
1813 y := 2;
```

```
1814 Echo(x, "= ", y, ":", x = y);
```

```
1815 Echo(x, "!= ", y, ":", x != y);
```

```
1816 Echo(x, "< ", y, ":", x < y);
```

```
1817 Echo(x, "<= ", y, ":", x <= y);
```

```
1818 Echo(x, "> ", y, ":", x > y);
```

```
1819 Echo(x, ">= ", y, ":", x >= y);
```

```
1820 %/mathpiper
```

```
1821     %output,preserve="false"
```

```
1822     Result: True
```

```
1823
```

```
1824     Side Effects:
```

```
1825     3 = 2 :False
```

```
1826     3 != 2 :True
```

```
1827     3 < 2 :False
```

```
1828     3 <= 2 :False
```

```
1829     3 > 2 :True
```

```
1830     3 >= 2 :True
```

```
1831 .    %/output
```

```
1832 Conditional operators are placed at a lower level of precedence than the other
1833 operators we have covered to this point:
```

```
1834     ()    Parentheses are evaluated from the inside out.
```

```
1835     ^     Then exponents are evaluated right to left.
```

1836      \*,%,/ Then multiplication, remainder, and division operations are evaluated  
1837              left to right.

1838      +, - Then addition and subtraction are evaluated left to right.

1839      =,!=,<,<=,>,>= Finally, conditional operators are evaluated.

## 1840    **12.2 Predicate Expressions**

1841    Expressions which return either **True** or **False** are called "**predicate**"  
1842    expressions. By themselves, predicate expressions are not very useful and they  
1843    only become so when they are used with special decision making functions, like  
1844    the If() function (which is discussed in the next section).

## 1845    **12.3 Exercises**

1846    For the following exercises, create a new MathRider worksheet file called  
1847    **book\_1\_section\_12a\_exercises\_<your first name>\_<your last name>.mrw.**  
1848    (**Note: there are no spaces in this file name**). For example, John Smith's  
1849    worksheet would be called:

1850    **book\_1\_section\_12a\_exercises\_john\_smith.mrw.**

1851    After this worksheet has been created, place your answer for each exercise that  
1852    requires a fold into its own fold in this worksheet. Place a title attribute in the  
1853    start tag of each fold which indicates the exercise the fold contains the solution  
1854    to. The folds you create should look similar to this one:

1855    `%mathpiper,title="Exercise 1"`

1856    `//Sample fold.`

1857    `%/mathpiper`

1858    If an exercise uses the MathPiper console instead of a fold, copy the work you  
1859    did in the console into the worksheet so it can be saved but do not put it in a fold.

### 1860    **12.3.1 Exercise 1**

1861    Carefully read all of section 12 up to this point. Evaluate each one of  
1862    the examples in the sections you read in the MathPiper worksheet you  
1863    created or in the MathPiper console and verify that the results match the  
1864    ones in the book. Copy all of the console examples you evaluated into your  
1865    worksheet so they will be saved but do not put them in a fold.

### 1866 **12.3.2 Exercise 2**

1867 Open a MathPiper session and evaluate the following predicate expressions:

1868 `In> 3 = 3`

1869 `In> 3 = 4`

1870 `In> 3 < 4`

1871 `In> 3 != 4`

1872 `In> -3 < 4`

1873 `In> 4 >= 4`

1874 `In> 1/2 < 1/4`

1875 `In> 15/23 < 122/189`

1876 `/*In the following two expressions, notice that 1/2 is not considered to be`  
1877 `equal to .5 unless it is converted to a numerical value first.*/`

1878 `In> 1/2 = .5`

1879 `In> N(1/2) = .5`

### 1880 **12.3.3 Exercise 3**

1881 Come up with 10 predicate expressions of your own and evaluate them in the  
1882 MathPiper console.

## 1883 **12.4 Making Decisions With The If() Function & Predicate Expressions**

1884 All programming languages have the ability to make decisions and the most  
1885 commonly used function for making decisions in MathPiper is the **If()** function.

1886 There are two calling formats for the If() function:

```
If(predicate, then)
If(predicate, then, else)
```

1887 The way the first form of the If() function works is that it evaluates the first  
1888 expression in its argument list (which is the "**predicate**" expression) and then  
1889 looks at the value that is returned. If this value is **True**, the "**then**" expression  
1890 that is listed second in the argument list is executed. If the predicate expression  
1891 evaluates to **False**, the "**then**" expression is not executed. (Note: any function

1892 that accepts a predicate expression as a parameter can also accept the boolean  
1893 values True and False).

1894 The following program uses an **If()** function to determine if the value in variable  
1895 number is greater than 5. If number is greater than 5, the program will echo  
1896 "Greater" and then "End of program":

```
1897 %mathpiper
1898 number := 6;
1899 If(number > 5, Echo(number, "is greater than 5.));
1900 Echo("End of program.");
1901 %/mathpiper
1902     %output,preserve="false"
1903     Result: True
1904
1905     Side Effects:
1906     6 is greater than 5.
1907     End of program.
1908 .    %/output
```

1909 In this program, number has been set to 6 and therefore the expression number  
1910 > 5 is **True**. When the **If()** functions evaluates the **predicate expression** and  
1911 determines it is **True**, it then executes the **first Echo()** function. The **second**  
1912 **Echo()** function at the bottom of the program prints "End of program"  
1913 regardless of what the If() function does. (**Note: semicolons cannot be placed**  
1914 **after expressions which are in function calls.**)

1915 Here is the same program except that **number** has been set to **4** instead of **6**:

```
1916 %mathpiper
1917 number := 4;
1918 If(number > 5, Echo(number, "is greater than 5.));
1919 Echo("End of program.");
1920 %/mathpiper
1921     %output,preserve="false"
1922     Result: True
1923
1924     Side Effects:
1925     End of program.
1926 .    %/output
```

1927 This time the expression **number > 4** returns a value of **False** which causes the  
1928 **If()** function to not execute the "**then**" expression that was passed to it.

### 1929 12.4.1 If() Functions Which Include An "Else" Parameter

1930 The second form of the If() function takes a third "**else**" expression which is  
1931 executed only if the predicate expression is **False**. This program is similar to the  
1932 previous one except an "**else**" expression has been added to it:

```
1933 %mathpiper
1934 x := 4;
1935 If(x > 5, Echo(x, "is greater than 5."), Echo(x, "is NOT greater than 5.));
1936 Echo("End of program.");
1937 %/mathpiper
1938     %output, preserve="false"
1939     Result: True
1940
1941     Side Effects:
1942     4 is NOT greater than 5.
1943     End of program.
1944 .    %/output
```

### 1945 12.5 Exercises

1946 For the following exercises, create a new MathRider worksheet file called  
1947 **book\_1\_section\_12b\_exercises\_<your first name>\_<your last name>.mrw**.  
1948 (**Note: there are no spaces in this file name**). For example, John Smith's  
1949 worksheet would be called:

1950 **book\_1\_section\_12b\_exercises\_john\_smith.mrw**.

1951 After this worksheet has been created, place your answer for each exercise that  
1952 requires a fold into its own fold in this worksheet. Place a title attribute in the  
1953 start tag of each fold which indicates the exercise the fold contains the solution  
1954 to. The folds you create should look similar to this one:

```
1955 %mathpiper, title="Exercise 1"
1956 //Sample fold.
1957 %/mathpiper
```

1958 If an exercise uses the MathPiper console instead of a fold, copy the work you



1959 did in the console into the worksheet so it can be saved but do not put it in a fold.

### 1960 12.5.1 Exercise 1

1961 Carefully read all of section 12 starting at the end of the previous  
1962 exercises and up to this point. Evaluate each one of the examples in the  
1963 sections you read in the MathPiper worksheet you created or in the  
1964 MathPiper console and verify that the results match the ones in the book.  
1965 Copy all of the console examples you evaluated into your worksheet so they  
1966 will be saved but do not put them in a fold.

### 1967 12.5.2 Exercise 2

1968 Write a program which uses the RandomInteger() function to simulate the  
1969 flipping of a coin (Hint: you can use 1 to represent a head and 0 to  
1970 represent a tail.). Use predicate expressions, the If() function, and the  
1971 Echo() function to print the string "**The coin came up heads.**" or the string  
1972 "**The coin came up tails.**", depending on what the simulated coin flip came  
1973 up as when the code was executed.

## 1974 12.6 The And(), Or(), & Not() Boolean Functions & Infix Notation

### 1975 12.6.1 And()

1976 Sometimes a programmer needs to check if two or more expressions are all **True**  
1977 and one way to do this is with the **And()** function. The And() function has **two**  
1978 **calling formats** (or **notations**) and this is the first one:

```
And(expression1, expression2, expression3, ..., expressionN)
```

1979 This calling format is able to accept one or more predicate expressions as input.  
1980 If **all** of these expressions returns a value of **True**, the And() function will also  
1981 return a **True**. However, if **any** of the expressions return a **False**, then the And()  
1982 function will return a **False**. This can be seen in the following example:

```
1983 In> And(True, True)  
1984 Result> True
```

```
1985 In> And(True, False)  
1986 Result> False
```

```
1987 In> And(False, True)  
1988 Result> False
```

```
1989 In> And(True, True, True, True)  
1990 Result> True
```

```
1991 In> And(True, True, False, True)
1992 Result> False
```

1993 The second format (or notation) that can be used to call the And() function is  
1994 called **infix** notation:

```
expression1 And expression2
```

1995 With **infix** notation, an expression is placed on both sides of the And() function  
1996 name instead of being placed inside of parentheses that are next to it:

```
1997 In> True And True
1998 Result> True
```

```
1999 In> True And False
2000 Result> False
```

```
2001 In> False And True
2002 Result> False
```

2003 Infix notation can only accept **two** expressions at a time, but it is often more  
2004 convenient to use than function calling notation. The following program also  
2005 demonstrates the infix version of the And() function being used:

```
2006 %mathpiper
```

```
2007 a := 7;
2008 b := 9;
```

```
2009 Echo("1: ", a < 5 And b < 10);
2010 Echo("2: ", a > 5 And b > 10);
2011 Echo("3: ", a < 5 And b > 10);
2012 Echo("4: ", a > 5 And b < 10);
```

```
2013 If(a > 5 And b < 10, Echo("These expressions are both true."));
```

```
2014 %/mathpiper
```

```
2015     %output,preserve="false"
2016     Result: True
2017
2018     Side Effects:
2019     1: False
2020     2: False
2021     3: False
2022     4: True
2023     These expressions are both true.
2024 .    %/output
```

2025 **12.6.2 Or()**

2026 The Or() function is similar to the And() function in that it has both a function  
2027 calling format and an infix calling format and it only works with predicate  
2028 expressions. However, instead of requiring that all expressions be **True** in order  
2029 to return a **True**, Or() will return a **True** if **one or more expressions are True**.

2030 Here is the function calling format for Or():

```
Or(expression1, expression2, expression3, ..., expressionN)
```

2031 and this example shows Or() being used with function calling format:

2032 In> Or(True, False)

2033 Result> True

2034 In> Or(False, True)

2035 Result> True

2036 In> Or(False, False)

2037 Result> False

2038 In> Or(False, False, False, False)

2039 Result> False

2040 In> Or(False, True, False, False)

2041 Result> True

2042 The infix notation format for Or() is as follows:

```
expression1 Or expression2
```

2043 and this example shows infix notation being used:

2044 In> True Or False

2045 Result> True

2046 In> False Or True

2047 Result> True

2048 In> False Or False

2049 Result> False

2050 The following program also demonstrates the infix version of the Or() function  
2051 being used:

```
2052 %mathpiper
2053 a := 7;
2054 b := 9;
2055 Echo("1: ", a < 5 Or b < 10);
2056 Echo("2: ", a > 5 Or b > 10);
2057 Echo("3: ", a > 5 Or b < 10);
2058 Echo("4: ", a < 5 Or b > 10);
2059 If(a < 5 Or b < 10, Echo("At least one of these expressions is true."));
2060 %/mathpiper
2061 %output,preserve="false"
2062 Result: True
2063
2064 Side Effects:
2065 1: True
2066 2: True
2067 3: True
2068 4: False
2069 At least one of these expressions is true.
2070 . %/output
```

### 2071 12.6.3 Not() & Prefix Notation

2072 The **Not()** function works with predicate expressions like the And() and Or()  
2073 functions do, except it can only accept **one** expression as input. The way Not()  
2074 works is that it changes a **True** value to a **False** value and a **False** value to a  
2075 **True** value. Here is the Not()'s function calling format:

```
Not(expression)
```

2076 and this example shows Not() being used with function calling format:

```
2077 In> Not(True)
2078 Result> False
2079 In> Not(False)
2080 Result> True
```

2081 Instead of providing an alternative infix calling format like And() and Or() do,  
2082 Not()'s second calling format uses **prefix** notation:

```
Not expression
```

2083 Prefix notation looks similar to function notation except no parentheses are used:

```
2084 In> Not True
2085 Result> False
```

```
2086 In> Not False
2087 Result> True
```

2088 Finally, here is a program that also uses the prefix version of Not():

```
2089 %mathpiper
2090 Echo("3 = 3 is ", 3 = 3);
2091 Echo("Not 3 = 3 is ", Not 3 = 3);
2092 %/mathpiper
2093     %output,preserve="false"
2094     Result: True
2095
2096     Side Effects:
2097     3 = 3 is True
2098     Not 3 = 3 is False
2099 .    %/output
```

## 2100 12.7 Exercises

2101 For the following exercises, create a new MathRider worksheet file called  
2102 **book\_1\_section\_12c\_exercises\_<your first name>\_<your last name>.mrw.**  
2103 (**Note: there are no spaces in this file name**). For example, John Smith's  
2104 worksheet would be called:

2105 **book\_1\_section\_12c\_exercises\_john\_smith.mrw.**

2106 After this worksheet has been created, place your answer for each exercise that  
2107 requires a fold into its own fold in this worksheet. Place a title attribute in the  
2108 start tag of each fold which indicates the exercise the fold contains the solution  
2109 to. The folds you create should look similar to this one:

```
2110 %mathpiper,title="Exercise 1"
2111 //Sample fold.
2112 %/mathpiper
```

2113 If an exercise uses the MathPiper console instead of a fold, copy the work you

2114 did in the console into the worksheet so it can be saved but do not put it in a fold.

### 2115 **12.7.1 Exercise 1**

2116 Carefully read all of section 12 starting at the end of the previous  
2117 exercises and up to this point. Evaluate each one of the examples in the  
2118 sections you read in the MathPiper worksheet you created or in the  
2119 MathPiper console and verify that the results match the ones in the book.  
2120 Copy all of the console examples you evaluated into your worksheet so they  
2121 will be saved but do not put them in a fold.

### 2122 **12.7.2 Exercise 2**

2123 The following program simulates the rolling of two dice and prints a  
2124 message if **both** of the two dice come up less than or equal to 3. Create a  
2125 similar program which simulates the flipping of two coins and print the  
2126 message "Both coins came up heads." if both coins come up heads.

```
2127 %mathpiper
2128 /*
2129     This program simulates the rolling of two dice and prints a message if
2130     both of the two dice come up less than or equal to 3.
2131 */
```

```
2132 die1 := RandomInteger(6);
2133 die2 := RandomInteger(6);

2134 Echo("Die1: ", die1, "   Die2: ", die2);
2135 NewLine();
```

```
2136 If( die1 <= 3 And die2 <= 3, Echo("Both dice came up <= to 3.") );
```

```
2137 %/mathpiper
```

### 2138 **12.7.3 Exercise 3**

2139 The following program simulates the rolling of two dice and prints a  
2140 message if **either** of the two dice come up less than or equal to 3. Create  
2141 a similar program which simulates the flipping of two coins and print the  
2142 message "At least one coin came up heads." if at least one coin comes up  
2143 heads.

```
2144 %mathpiper
2145 /*
2146     This program simulates the rolling of two dice and prints a message if
2147     either of the two dice come up less than or equal to 3.
2148 */
```

```
2149 die1 := RandomInteger(6);
2150 die2 := RandomInteger(6);
```

```
2151 Echo("Die1: ", die1, " Die2: ", die2);
2152 NewLine();

2153 If( die1 <= 3 Or die2 <= 3, Echo("At least one die came up <= 3.") );

2154 %/mathpiper
```

## 2155 13 The While() Looping Function & Bodied Notation

2156 Many kinds of machines, including computers, derive much of their power from  
2157 the principle of **repeated cycling**. **Repeated cycling** in a MathPiper program  
2158 means to execute one or more expressions over and over again and this process  
2159 is called "**looping**". MathPiper provides a number of ways to implement **loops**  
2160 in a program and these ways range from straight-forward to subtle.

2161 We will begin discussing looping in MathPiper by starting with the straight-  
2162 forward **While** function. The calling format for the **While** function is as follows:

```
2163 While(predicate)
2164 [
2165     body_expressions
2166 ];
```

2167 The **While** function is similar to the **If** function except it will repeatedly execute  
2168 the expressions it contains as long as its "predicate" expression is **True**. As soon  
2169 as the predicate expression returns a **False**, the While() function skips the  
2170 expressions it contains and execution continues with the expression that  
2171 immediately follows the While() function (if there is one).

2172 The expressions which are contained in a While() function are called its "**body**"  
2173 and all functions which have body expressions are called "**bodied**" functions. If  
2174 a body contains more than one expression then these expressions need to be  
2175 placed within a **code block** (code blocks were discussed in an earlier section).  
2176 What a function's body is will become clearer after studying some example  
2177 programs.

### 2178 13.1 Printing The Integers From 1 to 10

2179 The following program uses a While() function to print the integers from 1 to 10:

```
2180 %mathpiper
2181 // This program prints the integers from 1 to 10.
2182 /*
2183     Initialize the variable count to 1
2184     outside of the While "loop".
2185 */
2186 count := 1;
2187 While(count <= 10)
2188 [
2189     Echo(count);
```



```
2190
2191     count := count + 1; //Increment count by 1.
2192 ];
2193 %/mathpiper
2194     %output,preserve="false"
2195     Result: True
2196
2197     Side Effects:
2198     1
2199     2
2200     3
2201     4
2202     5
2203     6
2204     7
2205     8
2206     9
2207     10
2208 . %/output
```

2209 In this program, a single variable called **count** is created. It is used to tell the  
2210 Echo() function which integer to print and it is also used in the predicate  
2211 expression that determines if the While() function should continue to **loop** or not.

2212 When the program is executed, 1 is placed into **count** and then the While()  
2213 function is called. The predicate expression **count** <= 10 becomes **1** <= 10  
2214 and, since 1 is indeed less than or equal to 10, a value of **True** is returned by the  
2215 predicate expression.

2216 The While() function sees that the predicate expression returned a **True** and  
2217 therefore it executes all of the expressions inside of its **body** from top to bottom.

2218 The Echo() function prints the current contents of count (which is 1) and then the  
2219 expression count := count + 1 is executed.

2220 The expression **count := count + 1** is a standard expression form that is used in  
2221 many programming languages. Each time an expression in this form is  
2222 evaluated, it **increases the variable it contains by 1**. Another way to describe  
2223 the effect this expression has on **count** is to say that it **increments count by 1**.

2224 In this case **count** contains **1** and, after the expression is evaluated, **count**  
2225 contains **2**.

2226 After the last expression inside the body of the While() function is executed, the  
2227 While() function reevaluates its predicate expression to determine whether it  
2228 should continue looping or not. Since **count** is **2** at this point, the predicate  
2229 expression returns **True** and the code inside the body of the While() function is  
2230 executed again. This loop will be repeated until **count** is incremented to **11** and  
2231 the predicate expression returns **False**.

## 2232 **13.2 Printing The Integers From 1 to 100**

2233 The previous program can be adjusted in a number of ways to achieve different  
2234 results. For example, the following program prints the integers from 1 to 100 by  
2235 changing the **10** in the predicate expression to **100**. A Write() function is used in  
2236 this program so that its output is displayed on the same line until it encounters  
2237 the **wrap margin** in MathRider (which can be set in Utilities -> Buffer  
2238 Options...).

```
2239 %mathpiper
2240 // Print the integers from 1 to 100.
2241 count := 1;
2242 While(count <= 100)
2243 [
2244     Write(count,,);
2245     count := count + 1; //Increment count by 1.
2246 ];
2247
2248 %/mathpiper
2249     %output,preserve="false"
2250     Result: True
2251
2252     Side Effects:
2253     1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,
2254     24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,
2255     44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,
2256     64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,
2257     84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100
2258 . %/output
```

## 2259 **13.3 Printing The Odd Integers From 1 To 99**

2260 The following program prints the odd integers from 1 to 99 by changing the  
2261 **increment value** in the increment expression from **1** to **2**:

```
2262 %mathpiper
2263 //Print the odd integers from 1 to 99.
2264 x := 1;
2265 While(x <= 100)
2266 [
2267     Write(x,,);
```

```
2268     x := x + 2;    //Increment x by 2.
2269 ];

2270 %/mathpiper

2271     %output,preserve="false"
2272     Result: True
2273
2274     Side Effects:
2275     1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39,41,43,
2276     45,47,49,51,53,55,57,59,61,63,65,67,69,71,73,75,77,79,81,83,
2277     85,87,89,91,93,95,97,99
2278 .    %/output
```

### 2279 **13.4 Printing The Integers From 1 To 100 In Reverse Order**

2280 Finally, the following program prints the integers from 1 to 100 in reverse order:

```
2281 %mathpiper

2282 //Print the integers from 1 to 100 in reverse order.

2283 x := 100;

2284 While(x >= 1)
2285 [
2286     Write(x,,);
2287     x := x - 1;    //Decrement x by 1.
2288 ];

2289 %/mathpiper

2290     %output,preserve="false"
2291     Result: True
2292
2293     Side Effects:
2294     100,99,98,97,96,95,94,93,92,91,90,89,88,87,86,85,84,83,82,
2295     81,80,79,78,77,76,75,74,73,72,71,70,69,68,67,66,65,64,63,
2296     62,61,60,59,58,57,56,55,54,53,52,51,50,49,48,47,46,45,44,
2297     43,42,41,40,39,38,37,36,35,34,33,32,31,30,29,28,27,26,25,
2298     24,23,22,21,20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,
2299     3,2,1
2300 .    %/output
```

2301 In order to achieve the reverse ordering, this program had to initialize **x** to **100**,  
2302 check to see if **x** was **greater than or equal to 1** ( $x \geq 1$ ), and **decrement** **x** by  
2303 **subtracting 1 from it** instead of adding 1 to it.

### 2304 **13.5 Expressions Inside Of Code Blocks Are Indented**

2305 In the programs in the previous sections which use while loops, notice that the  
2306 expressions which are inside of the While() function's code block are **indented**.  
2307 These expressions do not need to be indented to execute properly, but doing so  
2308 makes the program easier to read.

### 2309 **13.6 Long-Running Loops, Infinite Loops, & Interrupting Execution**

2310 It is easy to create a loop that will execute a **large number of times**, or even **an**  
2311 **infinite number of times**, either on purpose or by mistake. When you execute  
2312 a program that contains an **infinite loop**, it will run until you tell MathPiper to  
2313 **interrupt** its execution. This is done by opening the MathPiper **console** and  
2314 then pressing the "**Halt Calculation**" button which in the upper left corner of  
2315 the console.

2316 Lets experiment with the **Halt Calculation** button by executing a program that  
2317 contains an infinite loop and then stopping it:

```
2318 %mathpiper
2319 //Infinite loop example program.
2320 x := 1;
2321 While(x < 10)
2322 [
2323     x := 3; //Oops, x is not being incremented!.
2324 ];
2325 %/mathpiper
2326     %output,preserve="false"
2327     Processing...
2328 .    %/output
```

2329 Since the contents of x is never changed inside the loop, the expression **x < 10**  
2330 always evaluates to **True** which causes the loop to continue looping. Notice that  
2331 the %output fold contains the word "**Processing...**" to indicate that the program  
2332 is still running the code.

2333 Execute this program now and then interrupt it using the **Halt Calculation**  
2334 button. When the program is interrupted, the %output fold will display the  
2335 message "**User interrupted calculation**" to indicate that the program was  
2336 interrupted. After a program has been interrupted, the program can be edited  
2337 and then rerun.

### 2338 **13.7 A Program That Simulates Rolling Two Dice 50 Times**

2339 The following program is larger than the previous programs that have been  
2340 discussed in this book, but it is also more interesting and more useful. It uses a  
2341 While() loop to simulate the rolling of two dice 50 times and it records how many  
2342 times each possible sum has been rolled so that this data can be printed. The  
2343 comments in the code explain what each part of the program does. (Remember, if  
2344 you copy this program to a MathRider worksheet, you can use **rectangular**  
2345 **selection mode** to easily remove the line numbers).

```
2346 %mathpiper
2347 /*
2348     This program simulates rolling two dice 50 times.
2349 */
2350 /*
2351     These variables are used to record how many times
2352     a possible sum of two dice has been rolled. They are
2353     all initialized to 0 before the simulation begins.
2354 */
2355 numberOfTwosRolled := 0;
2356 numberOfThreesRolled := 0;
2357 numberOfFoursRolled := 0;
2358 numberOfFivesRolled := 0;
2359 numberOfSixesRolled := 0;
2360 numberOfSevensRolled := 0;
2361 numberOfEightsRolled := 0;
2362 numberOfNinesRolled := 0;
2363 numberOfTensRolled := 0;
2364 numberOfElevensRolled := 0;
2365 numberOfTwelvesRolled := 0;
2366 //This variable keeps track of the number of the current roll.
2367 roll := 1;
2368 Echo("These are the rolls:");
2369 /*
2370     The simulation is performed inside of this while loop. The number of
2371     times the dice will be rolled can be changed by changing the number 50
2372     which is in the While function's predicate expression.
2373 */
2374 While(roll <= 50)
2375 [
2376     //Roll the dice.
2377     die1 := RandomInteger(6);
2378     die2 := RandomInteger(6);
```

```
2379
2380
2381 //Calculate the sum of the two dice.
2382 rollSum := die1 + die2;
2383
2384
2385 /*
2386 Print the sum that was rolled. Note: if a large number of rolls
2387 is going to be performed (say > 1000), it would be best to comment
2388 out this Write() function so that it does not put too much text
2389 into the output fold.
2390 */
2391 Write(rollSum,,);
2392
2393
2394 /*
2395 These If() functions determine which sum was rolled and then add
2396 1 to the variable which is keeping track of the number of times
2397 that sum was rolled.
2398 */
2399 If(rollSum = 2, numberOfTwosRolled := numberOfTwosRolled + 1);
2400 If(rollSum = 3, numberOfThreesRolled := numberOfThreesRolled + 1);
2401 If(rollSum = 4, numberOfFoursRolled := numberOfFoursRolled + 1);
2402 If(rollSum = 5, numberOfFivesRolled := numberOfFivesRolled + 1);
2403 If(rollSum = 6, numberOfSixesRolled := numberOfSixesRolled + 1);
2404 If(rollSum = 7, numberOfSevensRolled := numberOfSevensRolled + 1);
2405 If(rollSum = 8, numberOfEightsRolled := numberOfEightsRolled + 1);
2406 If(rollSum = 9, numberOfNinesRolled := numberOfNinesRolled + 1);
2407 If(rollSum = 10, numberOfTensRolled := numberOfTensRolled + 1);
2408 If(rollSum = 11, numberOfElevensRolled := numberOfElevensRolled+1);
2409 If(rollSum = 12, numberOfTwelvesRolled := numberOfTwelvesRolled+1);
2410
2411
2412 //Increment the roll variable to the next roll number.
2413 roll := roll + 1;
2414 ];
2415
2416 //Print the contents of the sum count variables for visual analysis.
2417 NewLine();
2418 NewLine();
2419 Echo("Number of Twos rolled: ", numberOfTwosRolled);
2420 Echo("Number of Threes rolled: ", numberOfThreesRolled);
2421 Echo("Number of Fours rolled: ", numberOfFoursRolled);
2422 Echo("Number of Fives rolled: ", numberOfFivesRolled);
2423 Echo("Number of Sixes rolled: ", numberOfSixesRolled);
2424 Echo("Number of Sevens rolled: ", numberOfSevensRolled);
2425 Echo("Number of Eights rolled: ", numberOfEightsRolled);
2426 Echo("Number of Nines rolled: ", numberOfNinesRolled);
2427 Echo("Number of Tens rolled: ", numberOfTensRolled);
2428 Echo("Number of Elevens rolled: ", numberOfElevensRolled);
2429 Echo("Number of Twelves rolled: ", numberOfTwelvesRolled);
```

```
2429 %/mathpiper
2430 %output,preserve="false"
2431 Result: True
2432
2433 Side effects:
2434 These are the rolls:
2435 4,8,6,4,6,9,7,11,9,3,11,6,11,7,11,4,7,7,8,7,3,6,7,7,7,12,4,
2436 12,7,8,12,6,8,10,10,5,9,8,4,5,3,5,7,7,4,6,7,7,5,8,
2437
2438 Number of Twos rolled: 0
2439 Number of Threes rolled: 3
2440 Number of Fours rolled: 6
2441 Number of Fives rolled: 4
2442 Number of Sixes rolled: 6
2443 Number of Sevens rolled: 13
2444 Number of Eights rolled: 6
2445 Number of Nines rolled: 3
2446 Number of Tens rolled: 2
2447 Number of Elevens rolled: 4
2448 Number of Twelves rolled: 3
2449 . %/output
```

## 2450 13.8 Exercises

2451 For the following exercises, create a new MathRider worksheet file called  
2452 **book\_1\_section\_13\_exercises\_<your first name>\_<your last name>.mrw.**  
2453 **(Note: there are no spaces in this file name).** For example, John Smith's  
2454 worksheet would be called:

2455 **book\_1\_section\_13\_exercises\_john\_smith.mrw.**

2456 After this worksheet has been created, place your answer for each exercise that  
2457 requires a fold into its own fold in this worksheet. Place a title attribute in the  
2458 start tag of each fold which indicates the exercise the fold contains the solution  
2459 to. The folds you create should look similar to this one:

```
2460 %mathpiper,title="Exercise 1"
```

```
2461 //Sample fold.
```

```
2462 %/mathpiper
```

2463 If an exercise uses the MathPiper console instead of a fold, copy the work you  
2464 did in the console into the worksheet so it can be saved but do not put it in a fold.

**2465 13.8.1 Exercise 1**

2466 Carefully read all of section 13 up to this point. Evaluate each one of  
2467 the examples in the sections you read in the MathPiper worksheet you  
2468 created or in the MathPiper console and verify that the results match the  
2469 ones in the book. Copy all of the console examples you evaluated into your  
2470 worksheet so they will be saved but do not put them in a fold.

**2471 13.8.2 Exercise 2**

2472 Create a program which uses a while loop to print the even integers from 2  
2473 to 50 inclusive.

**2474 13.8.3 Exercise 3**

2475 Create a program which prints all the multiples of 5 between 5 and 50  
2476 inclusive.

**2477 13.8.4 Exercise 4**

2478 Create a program which simulates the flipping of a coin 500 times. Print  
2479 the number of times the coin came up heads and the number of times it came  
2480 up tails after the loop is finished executing.



## 2481 **14 Predicate Functions**

2482 A **predicate function** is a function that either returns **True** or **False**. Most  
2483 predicate functions in MathPiper have names which begin with "**Is**". For  
2484 example, **IsEven()**, **IsOdd()**, **IsInteger()**, etc. The following examples show  
2485 some of the predicate functions that are in MathPiper:

```
2486 In> IsEven(4)
2487 Result> True
```

```
2488 In> IsEven(5)
2489 Result> False
```

```
2490 In> IsZero(0)
2491 Result> True
```

```
2492 In> IsZero(1)
2493 Result> False
```

```
2494 In> IsNegativeInteger(-1)
2495 Result> True
```

```
2496 In> IsNegativeInteger(1)
2497 Result> False
```

```
2498 In> IsPrime(7)
2499 Result> True
```

```
2500 In> IsPrime(100)
2501 Result> False
```

2502 There is also an **IsBound()** and an **IsUnbound()** function that can be used to  
2503 determine whether or not a value is bound to a given variable:

```
2504 In> a
2505 Result> a
```

```
2506 In> IsBound(a)
2507 Result> False
```

```
2508 In> a := 1
2509 Result> 1
```

```
2510 In> IsBound(a)
2511 Result> True
```

```
2512 In> Clear(a)
2513 Result> True
```

```
2514 In> a
2515 Result> a
```

```
2516 In> IsBound(a)
2517 Result> False
```

2518 The complete list of predicate functions is contained in the **User**  
2519 **Functions/Predicates** node in the MathPiperDocs plugin.

## 2520 **14.1 Finding Prime Numbers With A Loop**

2521 Predicate functions are very powerful when they are combined with loops  
2522 because they can be used to automatically make numerous checks. The  
2523 following program uses a while loop to pass the integers 1 through 20 (one at a  
2524 time) to the **IsPrime()** function in order to determine which integers are prime  
2525 and which integers are not prime:

```
2526 %mathpiper
2527 //Determine which numbers between 1 and 20 (inclusive) are prime.
2528 x := 1;
2529 While(x <= 20)
2530 [
2531     primeStatus := IsPrime(x);
2532     Echo(x, "is prime: ", primeStatus);
2533     x := x + 1;
2534 ];
2537 %/mathpiper
2538 %output,preserve="false"
2539 Result: True
2540
2541 Side Effects:
2542 1 is prime: False
2543 2 is prime: True
2544 3 is prime: True
2545 4 is prime: False
2546 5 is prime: True
2547 6 is prime: False
2548 7 is prime: True
2549 8 is prime: False
2550 9 is prime: False
2551 10 is prime: False
2552 11 is prime: True
2553 12 is prime: False
```

```
2554         13 is prime: True
2555         14 is prime: False
2556         15 is prime: False
2557         16 is prime: False
2558         17 is prime: True
2559         18 is prime: False
2560         19 is prime: True
2561         20 is prime: False
2562 .    %/output
```

2563 This program worked fairly well, but it is limited because it prints a line for each  
2564 prime number and also each non-prime number. This means that if large ranges  
2565 of integers were processed, enormous amounts of output would be produced.  
2566 The following program solves this problem by using an If() function to only print  
2567 a number if it is prime:

```
2568 %mathpiper
2569 //Print the prime numbers between 1 and 50 (inclusive).
2570 x := 1;
2571 While(x <= 50)
2572 [
2573     primeStatus := IsPrime(x);
2574     If(primeStatus = True, Write(x,,) );
2575     x := x + 1;
2576 ]
2577
2578 ];
2579 %/mathpiper
2580 %output,preserve="false"
2581     Result: True
2582
2583     Side Effects:
2584     2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,
2585 .    %/output
```

2586 This program is able to process a much larger range of numbers than the  
2587 previous one without having its output fill up the text area. However, the  
2588 program itself can be shortened by moving the **IsPrime()** function **inside** of the  
2589 **If()** function instead of using the **primeStatus** variable to communicate with it:

```
2590 %mathpiper
2591 /*
```

```
2592     Print the prime numbers between 1 and 50 (inclusive).
2593     This is a shorter version which places the IsPrime() function
2594     inside of the If() function instead of using a variable.
2595 */

2596 x := 1;

2597 While(x <= 50)
2598 [
2599     If(IsPrime(x), Write(x,,) );
2600     x := x + 1;
2601 ];

2602 ];

2603 %/mathpiper

2604 %output,preserve="false"
2605     Result: True
2606
2607     Side Effects:
2608     2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,
2609     . %/output
```

## 2610 **14.2 Finding The Length Of A String With The Length() Function**

2611 Strings can contain zero or more characters and the **Length()** function can be  
2612 used to determine how many characters a string holds:

```
2613 In> s := "Red"
2614 Result> "Red"

2615 In> Length(s)
2616 Result> 3
```

2617 In this example, the string "Red" is assigned to the variable **s** and then **s** is  
2618 passed to the **Length()** function. The **Length()** function returned a **3** which  
2619 means the string contained **3 characters**.

2620 The following example shows that strings can also be passed to functions  
2621 directly:

```
2622 In> Length("Red")
2623 Result> 3
```

2624 An **empty string** is represented by **two double quote marks with no space in**  
2625 **between them**. The **length** of an empty string is **0**:

```
2626 In> Length("")
2627 Result> 0
```

### 2628 **14.3 Converting Numbers To Strings With The String() Function**

2629 Sometimes it is useful to convert a number to a string so that the individual  
2630 digits in the number can be analyzed or manipulated. The following example  
2631 shows a **number** being converted to a **string** with the **String()** function so that  
2632 its **leftmost** and **rightmost** digits can be placed into **variables**:

```
2633 In> number := 523
2634 Result> 523
```

```
2635 In> stringNumber := String(number)
2636 Result> "523"
```

```
2637 In> leftmostDigit := stringNumber[1]
2638 Result> "5"
```

```
2639 In> rightmostDigit := stringNumber[ Length(stringNumber) ]
2640 Result> "3"
```

2641 Notice that the Length() function is used here to determine which character in  
2642 **stringNumber** held the **rightmost** digit.

### 2643 **14.4 Finding Prime Numbers Which End With 7 (And Multi-line Function** 2644 **Calls)**

2645 Now that we have covered how to turn a number into a string, lets use this  
2646 ability inside a loop. The following program finds all the **prime numbers**  
2647 between **1** and **500** which have a **7 as their rightmost digit**. There are three  
2648 important things which are shown in this program:

2649 1) Function calls **can have their parameters placed on more than one**  
2650 **line** if the parameters are too long to fit on a **single line**. In this case, a long  
2651 code block is being placed inside of an If() function.

2652 2) Code blocks (which are considered to be compound expressions) **cannot**  
2653 **have a semicolon placed after them if they are in a function call**. If a  
2654 semicolon is placed after this code block, an error will be produced.

2655 3) If() functions can be placed inside of other If() functions in order to make  
2656 more complex decisions. This is referred to as **nesting** functions.

2657 When the program is executed, it finds 24 prime numbers which have 7 as their  
2658 rightmost digit:

```
2659 %mathpiper
2660 /*
2661      Find all the prime numbers between 1 and 500 which have a 7
2662      as their rightmost digit.
2663 */
2664 x := 1;
2665 While(x <= 500)
2666 [
2667     //Notice how function parameters can be put on more than one line.
2668     If(IsPrime(x),
2669         [
2670             stringVersionOfNumber := String(x);
2671             stringLength := Length(stringVersionOfNumber);
2672             //Notice that If() functions can be placed inside of other
2673             // If() functions.
2674             If(stringVersionOfNumber[stringLength] = "7", Write(x,,) );
2675         ] //Notice that semicolons cannot be placed after code blocks
2676         //which are in function calls.
2677     ); //This is the close parentheses for the outer If() function.
2678     x := x + 1;
2679 ];
2680
2681 %/mathpiper
2682
2683 %output,preserve="false"
2684 Result: True
2685
2686 Side Effects:
2687 7,17,37,47,67,97,107,127,137,157,167,197,227,257,277,307,317,
2688 337,347,367,397,457,467,487,
2689 . %/output
```

2693 It would be nice if we had the ability to store these numbers someplace so that  
2694 they could be processed further and this is discussed in the next section.

## 2695 14.5 Exercises

2696 For the following exercises, create a new MathRider worksheet file called  
2697 **book\_1\_section\_14\_exercises\_<your first name>\_<your last name>.mrw.**  
2698 (**Note: there are no spaces in this file name**). For example, John Smith's  
2699 worksheet would be called:

**2700 book\_1\_section\_14\_exercises\_john\_smith.mrw.**

2701 After this worksheet has been created, place your answer for each exercise that  
2702 requires a fold into its own fold in this worksheet. Place a title attribute in the  
2703 start tag of each fold which indicates the exercise the fold contains the solution  
2704 to. The folds you create should look similar to this one:

```
2705 %mathpiper,title="Exercise 1"
```

```
2706 //Sample fold.
```

```
2707 %/mathpiper
```

2708 If an exercise uses the MathPiper console instead of a fold, copy the work you  
2709 did in the console into the worksheet so it can be saved but do not put it in a fold.

**2710 14.5.1 Exercise 1**

2711 Carefully read all of section 14 up to this point. Evaluate each one of  
2712 the examples in the sections you read in the MathPiper worksheet you  
2713 created or in the MathPiper console and verify that the results match the  
2714 ones in the book. Copy all of the console examples you evaluated into your  
2715 worksheet so they will be saved but do not put them in a fold.

**2716 14.5.2 Exercise 2**

2717 Write a program which uses a loop to determine how many prime numbers there  
2718 are between 1 and 1000. You do not need to print the numbers themselves,  
2719 just how many there are.

**2720 14.5.3 Exercise 3**

2721 Write a program which uses a loop to print all of the prime numbers between  
2722 10 and 99 which contain the digit 3 in either their 1's place, or their  
2723 10's place, or both places.

## 2724 15 Lists: Values That Hold Sequences Of Expressions

2725 The **list** value type is designed to hold expressions in an **ordered collection** or  
2726 **sequence**. Lists are very flexible and they are one of the most heavily used  
2727 value types in MathPiper. Lists can **hold expressions of any type**, they can be  
2728 made to **grow and shrink as needed**, and they can be **nested**. Expressions in a  
2729 list can be **accessed by their position** in the list (similar to the way that  
2730 characters in a string are accessed) and they can also be **replaced by other**  
2731 **expressions**.

2732 One way to create a list is by placing zero or more expressions separated by  
2733 commas inside of a **pair of braces {}**. In the following example, a list is created  
2734 that contains various expressions and then it is assigned to the variable **x**:

```
2735 In> x := {7,42,"Hello",1/2,var}  
2736 Result> {7,42,"Hello",1/2,var}
```

```
2737 In> x  
2738 Result> {7,42,"Hello",1/2,var}
```

2739 The number of expressions in a list can be determined with the **Length()**  
2740 function:

```
2741 In> Length({7,42,"Hello",1/2,var})  
2742 Result> 5
```

2743 A single expression in a list can be accessed by placing a set of **brackets []** to  
2744 the right of the variable that is bound to the list and then putting the  
2745 expression's position number inside of the brackets (**Note: the first expression**  
2746 **in the list is at position 1 counting from the left end of the list**):

```
2747 In> x[1]  
2748 Result> 7
```

```
2749 In> x[2]  
2750 Result> 42
```

```
2751 In> x[3]  
2752 Result> "Hello"
```

```
2753 In> x[4]  
2754 Result> 1/2
```

```
2755 In> x[5]  
2756 Result> var
```

2757 The **1st** and **2nd** expressions in this list are **integers**, the **3rd** expression is a



2758 **string**, the **4th** expression is a **rational number** and the **5th** expression is an  
2759 **unbound variable**.

2760 Lists can also hold other lists as shown in the following example:

```
2761 In> x := {20, 30, {31, 32, 33}, 40}
```

```
2762 Result> {20,30,{31,32,33},40}
```

```
2763 In> x[1]
```

```
2764 Result> 20
```

```
2765 In> x[2]
```

```
2766 Result> 30
```

```
2767 In> x[3]
```

```
2768 Result> {31,32,33}
```

```
2769 In> x[4]
```

```
2770 Result> 40
```

```
2771
```

2772 The expression in the **3rd** position in the list is another **list** which contains the  
2773 integers **31**, **32**, and **33**.

2774 An expression in this second list can be accessed by two **two sets of brackets**:

```
2775 In> x[3][2]
```

```
2776 Result> 32
```

2777 The **3** inside of the first set of brackets accesses the **3rd** member of the **first** list  
2778 and the **2** inside of the second set of brackets accesses the **2nd** member of the  
2779 **second** list.

## 2780 **15.1 Append() & Nondestructive List Operations**

```
Append(list, expression)
```

2781 The **Append()** function adds an expression to the end of a list:

```
2782 In> testList := {21,22,23}
```

```
2783 Result> {21,22,23}
```

```
2784 In> Append(testList, 24)
```

```
2785 Result> {21,22,23,24}
```

2786 However, instead of changing the **original** list, **Append()** creates a **copy** of the  
2787 **original** list and appends the expression to the **copy**. This can be confirmed by  
2788 evaluating the variable **testList** after the **Append()** function has been called:

```
2789 In> testList
2790 Result> {21,22,23}
```

2791 Notice that the list that is bound to **testList** was not modified by the **Append()**  
2792 function. This is called a **nondestructive list operation** and **most MathPiper**  
2793 **functions that manipulate lists do so nondestructively**. To have the new list  
2794 bound to the variable that is being used, the following technique can be  
2795 employed:

```
2796 In> testList := {21,22,23}
2797 Result> {21,22,23}

2798 In> testList := Append(testList, 24)
2799 Result> {21,22,23,24}
```

```
2800 In> testList
2801 Result> {21,22,23,24}
```

2802 After this code has been executed, the new list has indeed been bound to  
2803 **testList** as desired.

2804 There are some functions, such as **DestructiveAppend()**, which **do** change the  
2805 original list and most of them begin with the word "Destructive". These are  
2806 called "destructive functions" and they are advanced functions which are not  
2807 covered in this book.

## 2808 **15.2 Using While Loops With Lists**

2809 Functions that loop can be used to **select each expression in a list in turn** so  
2810 that an operation can be performed on these expressions. The following  
2811 program uses a while loop to print each of the expressions in a list:

```
2812 %mathpiper
2813 //Print each number in the list.

2814 x := {55,93,40,21,7,24,15,14,82};
2815 y := 1;

2816 While(y <= Length(x))
2817 [
2818     Echo(y, "- ", x[y]);
2819     y := y + 1;
2820 ];

2821 %/mathpiper
2822 %output,preserve="false"
```

```
2823         Result: True
2824
2825         Side Effects:
2826         1 - 55
2827         2 - 93
2828         3 - 40
2829         4 - 21
2830         5 - 7
2831         6 - 24
2832         7 - 15
2833         8 - 14
2834         9 - 82
2835     .    %/output
```

2836 A **loop** can also be used to search through a list. The following program uses a  
2837 **While()** function and an **If()** function to search through a list to see if it contains  
2838 the number **53**. If 53 is found in the list, a message is printed:

```
2839 %mathpiper

2840 //Determine if 53 is in the list.

2841 testList := {18,26,32,42,53,43,54,6,97,41};
2842 index := 1;

2843 While(index <= Length(testList))
2844 [
2845     If(testList[index] = 53,
2846         Echo("53 was found in the list at position", index));
2847     index := index + 1;
2848 ];

2850 %/mathpiper

2851 %output,preserve="false"
2852     Result: True
2853
2854     Side Effects:
2855     53 was found in the list at position 5
2856 .    %/output
```

2857 When this program was executed, it determined that **53** was present in the list at  
2858 position **5**.

### 2859 15.2.1 Using A While Loop And Append() To Place Values In A List

2860 In an earlier section it was mentioned that it would be nice if we could store a set  
2861 of values for later processing and this can be done with a **while loop** and the

2862 **Append()** function. The following program creates an empty list and assigned it  
2863 to the variable **primes**. The **while loop** and the **IsPrime()** function is then used  
2864 to locate the prime integers between 1 and 50 and the **Append()** function is used  
2865 to place them in the list. The last part of the program then prints some  
2866 information about the numbers that were placed into the list:

```
2867 %mathpiper
2868 //Place the prime numbers between 1 and 50 (inclusive) into a list.
2869 //Create an empty list.
2870 primes := {};
2871 x := 1;
2872 While(x <= 50)
2873 [
2874     /*
2875         If x is prime, append it to the end of the list and then assign
2876         the new list that is created to the variable 'primes'.
2877     */
2878     If(IsPrime(x), primes := Append(primes, x ) );
2879
2880     x := x + 1;
2881 ];
2882 //Print information about the primes that were found.
2883 Echo("Primes ", primes);
2884 Echo("The number of primes in the list = ", Length(primes) );
2885 Echo("The first number in the list = ", primes[1] );
2886 %/mathpiper
2887     %output,preserve="false"
2888     Result: True
2889
2890     Side Effects:
2891     Primes {2,3,5,7,11,13,17,19,23,29,31,37,41,43,47}
2892     The number of primes in the list = 15
2893     The first number in the list = 2
2894 .    %/output
```

2895 The ability to place values into a list with a loop is very powerful and we will be  
2896 using this ability throughout the rest of the book.

### 2897 15.3 Exercises

2898 For the following exercises, create a new MathRider worksheet file called  
2899 **book\_1\_section\_15a\_exercises\_<your first name>\_<your last name>.mrw.**

2900 **(Note: there are no spaces in this file name)**. For example, John Smith's  
2901 worksheet would be called:

2902 **book\_1\_section\_15a\_exercises\_john\_smith.mrw.**

2903 After this worksheet has been created, place your answer for each exercise that  
2904 requires a fold into its own fold in this worksheet. Place a title attribute in the  
2905 start tag of each fold which indicates the exercise the fold contains the solution  
2906 to. The folds you create should look similar to this one:

2907 `%mathpiper,title="Exercise 1"`

2908 `//Sample fold.`

2909 `%/mathpiper`

2910 If an exercise uses the MathPiper console instead of a fold, copy the work you  
2911 did in the console into the worksheet so it can be saved but do not put it in a fold.

### 2912 **15.3.1 Exercise 1**

2913 Carefully read all of section 15 up to this point. Evaluate each one of  
2914 the examples in the sections you read in the MathPiper worksheet you  
2915 created or in the MathPiper console and verify that the results match the  
2916 ones in the book. Copy all of the console examples you evaluated into your  
2917 worksheet so they will be saved but do not put them in a fold.

### 2918 **15.3.2 Exercise 2**

2919 Create a program that uses a loop and an IsOdd() function to analyze the  
2920 following list and then print the number of odd numbers it contains.

2921 `{73,94,80,37,56,94,40,21,7,24,15,14,82,93,32,74,22,68,65,52,85,61,46,86,25}`

### 2922 **15.3.3 Exercise 3**

2923 Create a program that uses a loop and an IsNegativeNumber() function to  
2924 copy all of the negative numbers in the following list into a new list.  
2925 Use the variable **negativeNumbers** to hold the new list.

2926 `{36,-29,-33,-6,14,7,-16,-3,-14,37,-38,-8,-45,-21,-26,6,6,38,-20,33,41,-`  
2927 `4,24,37,40,29}`

### 2928 **15.3.4 Exercise 4**

2929 Create a program that uses a loop to analyze the following list and then  
2930 print the following information about it:

- 2931 1) The largest number in the list.  
2932 2) The smallest number in the list.  
2933 3) The sum of all the numbers in the list.

```
2934 {73,94,80,37,56,94,40,21,7,24,15,14,82,93,32,74,22,68,65,52,85,61,46,86,25}
```

## 2935 **15.4 The ForEach() Looping Function**

2936 The **ForEach()** function uses a **loop** to index through a list like the While()  
2937 function does, but it is more flexible and automatic. ForEach() also uses bodied  
2938 notation like the While() function and here is its calling format:

```
ForEach(variable, list) body
```

2939 **ForEach()** selects each expression in a list in turn, assigns it to the passed-in  
2940 "variable", and then executes the expressions that are inside of "body".  
2941 Therefore, body is **executed once for each expression in the list**.

## 2942 **15.5 Print All The Values In A List Using A ForEach() function**

2943 This example shows how ForEach() can be used to print all of the items in a list:

```
2944 %mathpiper
```

```
2945 //Print all values in a list.
```

```
2946 ForEach(value, {50,51,52,53,54,55,56,57,58,59})
```

```
2947 [  
2948     Echo(value);  
2949 ];
```

```
2950 %/mathpiper
```

```
2951     %output,preserve="false"
```

```
2952     Result: True
```

```
2953  
2954     Side Effects:
```

```
2955     50
```

```
2956     51
```

```
2957     52
```

```
2958     53
```

```
2959     54
```

```
2960     55
```

```
2961     56
```

```
2962     57
```

```
2963     58
```

```
2964     59
```

```
2965 .    %/output
```

2966 **15.6 Calculate The Sum Of The Numbers In A List Using ForEach()**

2967 In previous examples, counting code in the form **x := x + 1** was used to count  
2968 how many times a while loop was executed. The following program uses a  
2969 **ForEach()** function and a line of code similar to this counter to calculate the  
2970 **sum of the numbers in a list:**

```
2971 %mathpiper
2972 /*
2973     This program calculates the sum of the numbers
2974     in a list.
2975 */
2976 //This variable is used to accumulate the sum.
2977 sum := 0;
2978 ForEach(x, {1,2,3,4,5,6,7,8,9,10} )
2979 [
2980     /*
2981         Add the contents of x to the contents of sum
2982         and place the result back into sum.
2983     */
2984     sum := sum + x;
2985
2986     //Print the sum as it is being accumulated.
2987     Write(sum,,);
2988 ];
2989 NewLine(); NewLine();
2990 Echo("The sum of the numbers in the list = ", sum);
2991 %/mathpiper
2992     %output,preserve="false"
2993     Result: True
2994
2995     Side Effects:
2996     1,3,6,10,15,21,28,36,45,55,
2997
2998     The sum of the numbers in the list = 55
2999 .    %/output
```

3000 In the above program, the integers **1** through **10** were manually placed into a list  
3001 by typing them individually. This method is limited because only a relatively  
3002 small number of integers can be placed into a list this way. The following section  
3003 discusses an operator which can be used to automatically place a large number  
3004 of integers into a list with very little typing.

## 3005 15.7 The .. Range Operator

```
first .. last
```

3006 A programmer often needs to create a list which contains **consecutive integers**  
3007 and the **.. "range"** operator can be used to do this. The **first** integer in the list is  
3008 placed before the **..** operator and the **last** integer in the list is placed after it  
3009 (**Note: there must be a space immediately to the left of the .. operator**  
3010 **and a space immediately to the right of it or an error will be generated.**).  
3011 Here are some examples:

```
3012 In> 1 .. 10  
3013 Result> {1,2,3,4,5,6,7,8,9,10}
```

```
3014 In> 10 .. 1  
3015 Result> {10,9,8,7,6,5,4,3,2,1}
```

```
3016 In> 1 .. 100  
3017 Result> {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,  
3018          21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,  
3019          38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,  
3020          55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,  
3021          72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,  
3022          89,90,91,92,93,94,95,96,97,98,99,100}
```

```
3023 In> -10 .. 10  
3024 Result> {-10,-9,-8,-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,8,9,10}
```

3025 As these examples show, the **..** operator can generate lists of integers in  
3026 ascending order and descending order. It can also generate lists that are very  
3027 large and ones that contain negative integers.

3028 Remember, though, if one or both of the spaces around the **..** are omitted, an  
3029 error is generated:

```
3030 In> 1..3  
3031 Result>  
3032 Error parsing expression, near token .3.
```

## 3033 15.8 Using ForEach() With The Range Operator To Print The Prime 3034 Numbers Between 1 And 100

3035 The following program shows how to use a **ForEach()** function instead of a  
3036 **While()** function to print the prime numbers between 1 and 100. Notice that  
3037 loops that are implemented with **ForEach()** often require less typing than  
3038 their **While()** based equivalents:



```
3039 %mathpiper
3040 /*
3041     This program prints the prime integers between 1 and 100 using
3042     a ForEach() function instead of a While() function. Notice that
3043     the ForEach() version requires less typing than the While()
3044     version.
3045 */
3046 ForEach(x, 1 .. 100)
3047 [
3048     If(IsPrime(x), Write(x,,) );
3049 ];
3050 %/mathpiper
3051     %output,preserve="false"
3052     Result: True
3053
3054     Side Effects:
3055     2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,
3056     73,79,83,89,97,
3057 .    %/output
```

### 3058 15.8.1 Using ForEach() And The Range Operator To Place The Prime 3059 Numbers Between 1 And 50 Into A List

3060 A ForEach() function can also be used to place values in a list, just the the  
3061 While() function can:

```
3062 %mathpiper
3063 /*
3064     Place the prime numbers between 1 and 50 into
3065     a list using a ForEach() function.
3066 */
3067 //Create a new list.
3068 primes := {};
3069 ForEach(number, 1 .. 50)
3070 [
3071     /*
3072         If number is prime, append it to the end of the list and
3073         then assign the new list that is created to the variable
3074         'primes'.
3075     */
3076     If(IsPrime(number), primes := Append(primes, number) );
3077 ];
```

```
3078 //Print information about the primes that were found.
3079 Echo("Primes ", primes);
3080 Echo("The number of primes in the list = ", Length(primes) );
3081 Echo("The first number in the list = ", primes[1] );

3082 %/mathpiper

3083     %output,preserve="false"
3084     Result: True
3085
3086     Side Effects:
3087     Primes {2,3,5,7,11,13,17,19,23,29,31,37,41,43,47}
3088     The number of primes in the list = 15
3089     The first number in the list = 2
3090 .    %/output
```

3091 As can be seen from the above examples, the **ForEach()** function and the **range**  
3092 **operator** can do a significant amount of work with very little typing. You will  
3093 discover in the next section that MathPiper has functions which are even more  
3094 powerful than these two.

## 3095 15.8.2 Exercises

3096 For the following exercises, create a new MathRider worksheet file called  
3097 **book\_1\_section\_15b\_exercises\_<your first name>\_<your last name>.mrw.**  
3098 **(Note: there are no spaces in this file name).** For example, John Smith's  
3099 worksheet would be called:

3100 **book\_1\_section\_15b\_exercises\_john\_smith.mrw.**

3101 After this worksheet has been created, place your answer for each exercise that  
3102 requires a fold into its own fold in this worksheet. Place a title attribute in the  
3103 start tag of each fold which indicates the exercise the fold contains the solution  
3104 to. The folds you create should look similar to this one:

```
3105 %mathpiper,title="Exercise 1"

3106 //Sample fold.

3107 %/mathpiper
```

3108 If an exercise uses the MathPiper console instead of a fold, copy the work you  
3109 did in the console into the worksheet so it can be saved but do not put it in a fold.

## 3110 15.8.3 Exercise 1

3111 Carefully read all of section 15 starting at the end of the previous  
3112 exercises and up to this point. Evaluate each one of the examples in the

3113 sections you read in the MathPiper worksheet you created or in the  
3114 MathPiper console and verify that the results match the ones in the book.  
3115 Copy all of the console examples you evaluated into your worksheet so they  
3116 will be saved but do not put them in a fold.

## 3117 **15.8.4 Exercise 2**

3118 Create a program that uses a **ForEach()** function and an **IsOdd()** function to  
3119 analyze the following list and then print the number of odd numbers it  
3120 contains.

3121 {73,94,80,37,56,94,40,21,7,24,15,14,82,93,32,74,22,68,65,52,85,61,46,86,25}

## 3122 **15.8.5 Exercise 3**

3123 Create a program that uses a **ForEach()** function and an **IsNegativeNumber()**  
3124 function to copy all of the negative numbers in the following list into a  
3125 new list. Use the variable **negativeNumbers** to hold the new list.

3126 {36,-29,-33,-6,14,7,-16,-3,-14,37,-38,-8,-45,-21,-26,6,6,38,-20,33,41,-  
3127 4,24,37,40,29}

## 3128 **15.8.6 Exercise 4**

3129 Create a program that uses a **ForEach()** function to analyze the following  
3130 list and then print the following information about it:

- 3131 1) The largest number in the list.  
3132 2) The smallest number in the list.  
3133 3) The sum of all the numbers in the list.

3134 {73,94,80,37,56,94,40,21,7,24,15,14,82,93,32,74,22,68,65,52,85,61,46,86,25}

## 3135 **15.8.7 Exercise 5**

3136 Create a program that uses a **while loop** to make a list that contains **1000**  
3137 **random integers** between **1** and **100** inclusive. Then, use a **ForEach()**  
3138 function to determine how many integers in the list are **prime** and use an  
3139 **Echo()** function to print this total.

## 3140 **16 Functions & Operators Which Loop Internally**

3141 Looping is such a useful capability that MathPiper has many functions which  
3142 loop internally. Now that you have some experience with loops, you can use this  
3143 experience to help you imagine how these functions use loops to process the  
3144 information that is passed to them.

### 3145 **16.1 Functions & Operators Which Loop Internally To Process Lists**

3146 This section discusses a number of functions that use loops to process lists.

#### 3147 **16.1.1 TableForm()**

```
TableForm(list)
```

3148 The **TableForm()** function prints the contents of a list in the form of a table.  
3149 Each member in the list is printed on its own line and this sometimes makes the  
3150 contents of the list easier to read:

```
3151 In> testList := {2,4,6,8,10,12,14,16,18,20}  
3152 Result> {2,4,6,8,10,12,14,16,18,20}
```

```
3153 In> TableForm(testList)  
3154 Result> True  
3155 Side Effects>  
3156 2  
3157 4  
3158 6  
3159 8  
3160 10  
3161 12  
3162 14  
3163 16  
3164 18  
3165 20
```

#### 3166 **16.1.2 Contains()**

3167 The **Contains()** function searches a list to determine if it contains a given  
3168 expression. If it finds the expression, it returns **True** and if it doesn't find the  
3169 expression, it returns **False**. Here is the calling format for Contains():

```
Contains(list, expression)
```

3170 The following code shows Contains() being used to locate a number in a list:

```
3171 In> Contains({50,51,52,53,54,55,56,57,58,59}, 53)
3172 Result> True
```

```
3173 In> Contains({50,51,52,53,54,55,56,57,58,59}, 75)
3174 Result> False
```

3175 The **Not()** function can also be used with predicate functions like Contains() to  
3176 change their results to the opposite truth value:

```
3177 In> Not Contains({50,51,52,53,54,55,56,57,58,59}, 75)
3178 Result> True
```

### 3179 16.1.3 Find()

```
Find(list, expression)
```

3180 The **Find()** function searches a list for the first occurrence of a given expression.  
3181 If the expression is found, the **position of its first occurrence** is returned and  
3182 if it is not found, **-1** is returned:

```
3183 In> Find({23, 15, 67, 98, 64}, 15)
3184 Result> 2
```

```
3185 In> Find({23, 15, 67, 98, 64}, 8)
3186 Result> -1
```

### 3187 16.1.4 Count()

```
Count(list, expression)
```

3188 **Count()** determines the number of times a given expression occurs in a list:

```
3189 In> testList := {a,b,b,c,c,c,d,d,d,d,e,e,e,e,e}
3190 Result> {a,b,b,c,c,c,d,d,d,d,e,e,e,e,e}
```

```
3191 In> Count(testList, c)
3192 Result> 3
```

```
3193 In> Count(testList, e)
3194 Result> 5
```

```
3195 In> Count(testList, z)
3196 Result> 0
```

3197 **16.1.5 Select()**

```
Select(predicate function, list)
```

3198 **Select()** returns a list that contains all the expressions in a list which make a  
3199 given predicate function return **True**:

```
3200 In> Select("IsPositiveInteger", {46,87,59,-27,11,86,-21,-58,-86,-52})  
3201 Result> {46,87,59,11,86}
```

3202 In this example, notice that the **name** of the predicate function is passed to  
3203 Select() in **double quotes**. There are other ways to pass a predicate function to  
3204 Select() but these are covered in a later section.

3205 Here are some further examples which use the Select() function:

```
3206 In> Select("IsOdd", {16,14,82,92,33,74,99,67,65,52})  
3207 Result> {33,99,67,65}
```

```
3208 In> Select("IsEven", {16,14,82,92,33,74,99,67,65,52})  
3209 Result> {16,14,82,92,74,52}
```

```
3210 In> Select("IsPrime", 1 .. 75)  
3211 Result> {2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73}
```

3212 Notice how the third example uses the **..** operator to automatically generate a list  
3213 of consecutive integers from 1 to 75 for the Select() function to analyze.

3214 **16.1.6 The Nth() Function & The [] Operator**

```
Nth(list, index)
```

3215 The **Nth()** function simply returns the expression which is at a given position in  
3216 a list. This example shows the **third** expression in a list being obtained:

```
3217 In> testList := {a,b,c,d,e,f,g}  
3218 Result> {a,b,c,d,e,f,g}
```

```
3219 In> Nth(testList, 3)  
3220 Result> c
```

3221 As discussed earlier, the **[]** operator can also be used to obtain a single  
3222 expression from a list:

```
3223 In> testList[3]
3224 Result> c
```

3225 The **[]** operator can even obtain a single expression directly from a list without  
3226 needing to use a variable:

```
3227 In> {a,b,c,d,e,f,g}[3]
3228 Result> c
```

### 3229 16.1.7 The : Prepend Operator

```
expression : list
```

3230 The prepend operator is a colon **:** and it can be used to add an expression to the  
3231 beginning of a list:

```
3232 In> testList := {b,c,d}
3233 Result> {b,c,d}

3234 In> testList := a:testList
3235 Result> {a,b,c,d}
```

### 3236 16.1.8 Concat()

```
Concat(list1, list2, ...)
```

3237 The Concat() function is short for "concatenate" which means to join together  
3238 sequentially. It takes two or more lists and joins them together into a single  
3239 larger list:

```
3240 In> Concat({a,b,c}, {1,2,3}, {x,y,z})
3241 Result> {a,b,c,1,2,3,x,y,z}
```

### 3242 16.1.9 Insert(), Delete(), & Replace()

```
Insert(list, index, expression)
```

```
Delete(list, index)
```

```
Replace(list, index, expression)
```

3243 **Insert()** inserts an expression into a list at a given index, **Delete()** deletes an  
3244 expression from a list at a given index, and **Replace()** replaces an expression in  
3245 a list at a given index with another expression:

```
3246 In> testList := {a,b,c,d,e,f,g}
3247 Result> {a,b,c,d,e,f,g}

3248 In> testList := Insert(testList, 4, 123)
3249 Result> {a,b,c,123,d,e,f,g}

3250 In> testList := Delete(testList, 4)
3251 Result> {a,b,c,d,e,f,g}

3252 In> testList := Replace(testList, 4, xxx)
3253 Result> {a,b,c,xxx,e,f,g}
```

#### 3254 16.1.10 Take()

```
Take(list, amount)
Take(list, -amount)
Take(list, {begin_index,end_index})
```

3255 **Take()** obtains a sublist from the **beginning** of a list, the **end** of a list, or the  
3256 **middle** of a list. The expressions in the list that are not taken are discarded.

3257 A **positive** integer passed to Take() indicates how many expressions should be  
3258 taken from the **beginning** of a list:

```
3259 In> testList := {a,b,c,d,e,f,g}
3260 Result> {a,b,c,d,e,f,g}

3261 In> Take(testList, 3)
3262 Result> {a,b,c}
```

3263 A **negative** integer passed to Take() indicates how many expressions should be  
3264 taken from the **end** of a list:

```
3265 In> Take(testList, -3)
3266 Result> {e,f,g}
```

3267 Finally, if a **two member list** is passed to Take() it indicates the **range** of  
3268 expressions that should be taken from the **middle** of a list. The **first** value in the  
3269 passed-in list specifies the **beginning** index of the range and the **second** value  
3270 specifies its **end**:

```
3271 In> Take(testList, {3,5})
3272 Result> {c,d,e}
```



3273 **16.1.11 Drop()**

```
Drop(list, index)
Drop(list, -index)
Drop(list, {begin_index,end_index})
```

3274 **Drop()** does the opposite of Take() in that it **drops** expressions from the  
3275 **beginning** of a list, the **end** of a list, or the **middle** of a list and **returns a list**  
3276 **which contains the remaining expressions.**

3277 A **positive** integer passed to Drop() indicates how many expressions should be  
3278 dropped from the **beginning** of a list:

```
3279 In> testList := {a,b,c,d,e,f,g}
3280 Result> {a,b,c,d,e,f,g}
```

```
3281 In> Drop(testList, 3)
3282 Result> {d,e,f,g}
```

3283 A **negative** integer passed to Drop() indicates how many expressions should be  
3284 dropped from the **end** of a list:

```
3285 In> Drop(testList, -3)
3286 Result> {a,b,c,d}
```

3287 Finally, if a **two member list** is passed to Drop() it indicates the **range** of  
3288 expressions that should be dropped from the **middle** of a list. The **first** value in  
3289 the passed-in list specifies the **beginning** index of the range and the **second**  
3290 value specifies its **end**:

```
3291 In> Drop(testList, {3,5})
3292 Result> {a,b,f,g}
```

3293 **16.1.12 FillList()**

```
FillList(expression, length)
```

3294 The FillList() function simply creates a list which is of size "length" and fills it  
3295 with "length" copies of the given expression:

```
3296 In> FillList(a, 5)
3297 Result> {a,a,a,a,a}
```

```
3298 In> FillList(42,8)
3299 Result> {42,42,42,42,42,42,42,42}
```

**3300 16.1.13 RemoveDuplicates()**

```
RemoveDuplicates(list)
```

**3301 RemoveDuplicates()** removes any duplicate expressions that are contained in a  
3302 list:

3303 In> testList := {a,a,b,c,c,b,b,a,b,c,c}

3304 Result> {a,a,b,c,c,b,b,a,b,c,c}

3305 In> RemoveDuplicates(testList)

3306 Result> {a,b,c}

**3307 16.1.14 Reverse()**

```
Reverse(list)
```

**3308 Reverse()** reverses the order of the expressions in a list:

3309 In> testList := {a,b,c,d,e,f,g,h}

3310 Result> {a,b,c,d,e,f,g,h}

3311 In> Reverse(testList)

3312 Result> {h,g,f,e,d,c,b,a}

**3313 16.1.15 Partition()**

```
Partition(list, partition_size)
```

**3314 The Partition()** function breaks a list into sublists of size "partition\_size":

3315 In> testList := {a,b,c,d,e,f,g,h}

3316 Result> {a,b,c,d,e,f,g,h}

3317 In> Partition(testList, 2)

3318 Result> {{a,b},{c,d},{e,f},{g,h}}

**3319 If the partition\_size does not divide the length of the list *evenly*, the remaining**  
**3320 elements are discarded:**

3321 In> Partition(testList, 3)

3322 Result> {{h,b,c},{d,e,f}}

3323 The number of elements that Partition() will discard can be calculated by  
3324 dividing the length of a list by the partition size and obtaining the **remainder**:

```
3325 In> Length(testList) % 3  
3326 Result> 2
```

3327 Remember that % is the remainder operator. It divides two integers and returns  
3328 their remainder.

### 3329 **16.1.16 Table()**

```
Table(expression, variable, begin_value, end_value, step_amount)
```

3330 The Table() function creates a list of values by doing the following:

- 3331 1) Generating a sequence of values between a "begin\_value" and an  
3332 "end\_value" with each value being incremented by the "step\_amount".
- 3333 2) Placing each value in the sequence into the specified "variable", one value  
3334 at a time.
- 3335 3) Evaluating the defined "expression" (which contains the defined "variable")  
3336 for each value, one at a time.
- 3337 4) Placing the result of each "expression" evaluation into the result list.

3338 This example generates a list which contains the integers 1 through 10:

```
3339 In> Table(x, x, 1, 10, 1)  
3340 Result> {1,2,3,4,5,6,7,8,9,10}
```

3341 Notice that the expression in this example is simply the variable 'x' itself with no  
3342 other operations performed on it.

3343 The following example is similar to the previous one except that its expression  
3344 multiplies 'x' by 2:

```
3345 In> Table(x*2, x, 1, 10, 1)  
3346 Result> {2,4,6,8,10,12,14,16,18,20}
```

3347 Lists which contain decimal values can also be created by setting the  
3348 "step\_amount" to a decimal:

```
3349 In> Table(x, x, 0, 1, .1)  
3350 Result> {0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1}
```

3351 **16.1.17 HeapSort()**

```
HeapSort(list, compare)
```

3352 **HeapSort()** sorts the elements of **list** into the order indicated by **compare** with  
3353 **compare** typically being the **less than** operator "<" or the **greater than**  
3354 operator ">":

```
3355 In> HeapSort({4,7,23,53,-2,1}, "<");  
3356 Result: {-2,1,4,7,23,53}
```

```
3357 In> HeapSort({4,7,23,53,-2,1}, ">");  
3358 Result: {53,23,7,4,1,-2}
```

```
3359 In> HeapSort({1/2,3/5,7/8,5/16,3/32}, "<")  
3360 Result: {3/32,5/16,1/2,3/5,7/8}
```

```
3361 In> HeapSort({.5,3/5,.76,5/16,3/32}, "<")  
3362 Result: {3/32,5/16,.5,3/5,.76}
```

3363 **16.2 Functions That Work With Integers**

3364 This section discusses various functions which work with integers. Some of  
3365 these functions also work with non-integer values and their use with non-  
3366 integers is discussed in other sections.

3367 **16.2.1 RandomIntegerVector()**

```
RandomIntegerVector(length, lowest_possible, highest_possible)
```

3368 A vector is a list that does not contain other lists. **RandomIntegerVector()**  
3369 creates a list of size "length" that contains random integers that are no lower  
3370 than "lowest\_possible" and no higher than "highest possible". The following  
3371 example creates **10** random integers between **1** and **99** inclusive:

```
3372 In> RandomIntegerVector(10, 1, 99)  
3373 Result> {73,93,80,37,55,93,40,21,7,24}
```

3374 **16.2.2 Max() & Min()**

```
Max(value1, value2)  
Max(list)
```

3375 If two values are passed to **Max()**, it determines which one is larger:

```
3376 In> Max(10, 20)
```

3377 `Result> 20`

3378 If a list of values are passed to `Max()`, it finds the largest value in the list:

3379 `In> testList := RandomIntegerVector(10, 1, 99)`

3380 `Result> {73,93,80,37,55,93,40,21,7,24}`

3381 `In> Max(testList)`

3382 `Result> 93`

3383 The **Min()** function is the opposite of the `Max()` function.

```
Min(value1, value2)
```

```
Min(list)
```

3384 If two values are passed to `Min()`, it determines which one is smaller:

3385 `In> Min(10, 20)`

3386 `Result> 10`

3387 If a list of values are passed to `Min()`, it finds the smallest value in the list:

3388 `In> testList := RandomIntegerVector(10, 1, 99)`

3389 `Result> {73,93,80,37,55,93,40,21,7,24}`

3390 `In> Min(testList)`

3391 `Result> 7`

### 3392 **16.2.3 Div() & Mod()**

```
Div(dividend, divisor)
```

```
Mod(dividend, divisor)
```

3393 **Div()** stands for "divide" and determines the whole number of times a divisor  
3394 goes into a dividend:

3395 `In> Div(7, 3)`

3396 `Result> 2`

3397 **Mod()** stands for "modulo" and it determines the remainder that results when a  
3398 dividend is divided by a divisor:

3399 `In> Mod(7,3)`

3400 `Result> 1`

3401 The remainder/modulo operator **%** can also be used to calculate a remainder:

```
3402 In> 7 % 2
3403 Result> 1
```

#### 3404 **16.2.4 Gcd()**

```
Gcd(value1, value2)
Gcd(list)
```

3405 GCD stands for Greatest Common Divisor and the **Gcd()** function determines the  
3406 greatest common divisor of the values that are passed to it.

3407 If two integers are passed to Gcd(), it calculates their greatest common divisor:

```
3408 In> Gcd(21, 56)
3409 Result> 7
```

3410 If a list of integers are passed to Gcd(), it finds the greatest common divisor of all  
3411 the integers in the list:

```
3412 In> Gcd({9, 66, 123})
3413 Result> 3
```

#### 3414 **16.2.5 Lcm()**

```
Lcm(value1, value2)
Lcm(list)
```

3415 LCM stands for Least Common Multiple and the **Lcm()** function determines the  
3416 least common multiple of the values that are passed to it.

3417 If two integers are passed to Lcm(), it calculates their least common multiple:

```
3418 In> Lcm(14, 8)
3419 Result> 56
```

3420 If a list of integers are passed to Lcm(), it finds the least common multiple of all  
3421 the integers in the list:

```
3422 In> Lcm({3, 7, 9, 11})
3423 Result> 693
```

3424 **16.2.6 Sum()**

```
Sum(list)
```

3425 **Sum()** can find the sum of a list that is passed to it:

3426 In&gt; testList := RandomIntegerVector(10,1,99)

3427 Result&gt; {73,93,80,37,55,93,40,21,7,24}

3428 In&gt; Sum(testList)

3429 Result&gt; 523

3430 In&gt; testList := 1 .. 10

3431 Result&gt; {1,2,3,4,5,6,7,8,9,10}

3432 In&gt; Sum(testList)

3433 Result&gt; 55

3434 **16.2.7 Product()**

```
Product(list)
```

3435 This function has two calling formats, only one of which is discussed here.

3436 **Product(list)** multiplies all the expressions in a list together and returns their  
3437 product:

3438 In&gt; Product({1,2,3})

3439 Result&gt; 6

3440 **16.3 Exercises**3441 For the following exercises, create a new MathRider worksheet file called  
3442 **book\_1\_section\_16\_exercises\_<your first name>\_<your last name>.mrw.**  
3443 **(Note: there are no spaces in this file name).** For example, John Smith's  
3444 worksheet would be called:3445 **book\_1\_section\_16\_exercises\_john\_smith.mrw.**3446 After this worksheet has been created, place your answer for each exercise that  
3447 requires a fold into its own fold in this worksheet. Place a title attribute in the  
3448 start tag of each fold which indicates the exercise the fold contains the solution  
3449 to. The folds you create should look similar to this one:

3450 %mathpiper,title="Exercise 1"

3451 //Sample fold.

3452 `%/mathpiper`

3453 If an exercise uses the MathPiper console instead of a fold, copy the work you  
3454 did in the console into the worksheet so it can be saved but do not put it in a fold.

### 3455 **16.3.1 Exercise 1**

3456 Carefully read all of section 16 up to this point. Evaluate each one of  
3457 the examples in the sections you read in the MathPiper worksheet you  
3458 created or in the MathPiper console and verify that the results match the  
3459 ones in the book. Copy all of the console examples you evaluated into your  
3460 worksheet so they will be saved but do not put them in a fold.

### 3461 **16.3.2 Exercise 2**

3462 Create a program that uses `RandomIntegerVector()` to create a 100 member  
3463 list that contains random integers between 1 and 5 inclusive. Use `Count()`  
3464 to determine how many of each digit 1-5 are in the list and then print this  
3465 information. Hint: you can use the `HeapSort()` function to sort the  
3466 generated list to make it easier to check if your program is counting  
3467 correctly.

### 3468 **16.3.3 Exercise 3**

3469 Create a program that uses `RandomIntegerVector()` to create a 100 member  
3470 list that contains random integers between 1 and 50 inclusive and use  
3471 `Contains()` to determine if the number 25 is in the list. Print "25 was in  
3472 the list." if 25 was found in the list and "25 was not in the list." if it  
3473 wasn't found.

### 3474 **16.3.4 Exercise 4**

3475 Create a program that uses `RandomIntegerVector()` to create a 100 member  
3476 list that contains random integers between 1 and 50 inclusive and use  
3477 `Find()` to determine if the number 10 is in the list. Print the position of  
3478 10 if it was found in the list and "10 was not in the list." if it wasn't  
3479 found.

### 3480 **16.3.5 Exercise 5**

3481 Create a program that uses `RandomIntegerVector()` to create a 100 member  
3482 list that contains random integers between 0 and 3 inclusive. Use `Select()`  
3483 with the `IsNonZeroInteger()` predicate function to obtain all of the nonzero  
3484 integers in this list.

### 3485 **16.3.6 Exercise 6**

3486 Create a program that uses `Table()` to obtain a list which contains the  
3487 squares of the integers between 1 and 10 inclusive.



## 3488 17 Nested Loops

3489 Now that you have seen how to solve problems with single loops, it is time to  
3490 discuss what can be done when a loop is placed inside of another loop. A loop  
3491 that is placed **inside** of another loop it is called a **nested loop** and this nesting  
3492 can be extended to numerous levels if needed. This means that loop 1 can have  
3493 loop 2 placed inside of it, loop 2 can have loop 3 placed inside of it, loop 3 can  
3494 have loop 4 placed inside of it, and so on.

3495 Nesting loops allows the programmer to accomplish an enormous amount of  
3496 work with very little typing.

### 3497 17.1 Generate All The Combinations That Can Be Entered Into A Two Digit 3498 Wheel Lock Using A Nested Loop



3499 The following program generates all the combinations that can be entered into a  
3500 two digit wheel lock. It uses a nested loop to accomplish this with the "**inside**"  
3501 nested loop being used to generate **one's place** digits and the "**outside**" loop  
3502 being used to generate **ten's place** digits.

```
3503 %mathpiper
3504 /*
3505  Generate all the combinations can be entered into a two
3506  digit wheel lock.
3507 */
3508 combinations := {};
3509 ForEach(digit1, 0 .. 9) //This loop is called the "outside" loop.
```

```
3510 [
3511     ForEach(digit2, 0 .. 9)//This loop is called the "inside" loop.
3512     [
3513         combinations := Append(combinations, {digit1, digit2});
3514     ];
3515 ];

3516 Echo(TableForm(combinations));

3517 %/mathpiper

3518     %output,preserve="false"
3519     Result: True
3520
3521     Side Effects:
3522     {0,0}
3523     {0,1}
3524     {0,2}
3525     {0,3}
3526     {0,4}
3527     {0,5}
3528     {0,6}
3529     .
3530     . //The middle of the list has not been shown.
3531     .
3532     {9,3}
3533     {9,4}
3534     {9,5}
3535     {9,6}
3536     {9,7}
3537     {9,8}
3538     {9,9}
3539     True
3540 . %/output
```

3541 The relationship between the outside loop and the inside loop is interesting  
3542 because each time the **outside loop cycles once**, the **inside loop cycles 10**  
3543 **times**. Study this program carefully because nested loops can be used to solve a  
3544 wide range of problems and therefore understanding how they work is  
3545 important.

## 3546 17.2 Exercises

3547 For the following exercises, create a new MathRider worksheet file called  
3548 **book\_1\_section\_17\_exercises\_<your first name>\_<your last name>.mrw**.  
3549 (**Note: there are no spaces in this file name**). For example, John Smith's  
3550 worksheet would be called:

3551 **book\_1\_section\_17\_exercises\_john\_smith.mrw**.

3552 After this worksheet has been created, place your answer for each exercise that  
3553 requires a fold into its own fold in this worksheet. Place a title attribute in the  
3554 start tag of each fold which indicates the exercise the fold contains the solution  
3555 to. The folds you create should look similar to this one:

3556 `%mathpiper,title="Exercise 1"`

3557 `//Sample fold.`

3558 `%/mathpiper`

3559 If an exercise uses the MathPiper console instead of a fold, copy the work you  
3560 did in the console into the worksheet so it can be saved but do not put it in a fold.

### 3561 **17.2.1 Exercise 1**

3562 Carefully read all of section 17 up to this point. Evaluate each one of  
3563 the examples in the sections you read in the MathPiper worksheet you  
3564 created or in the MathPiper console and verify that the results match the  
3565 ones in the book. Copy all of the console examples you evaluated into your  
3566 worksheet so they will be saved but do not put them in a fold.

### 3567 **17.2.2 Exercise 2**

3568 Create a program that will generate all of the combinations that can be  
3569 entered into a three digit wheel lock. (Hint: a triple nested loop can be  
3570 used to accomplish this.)

## 3571 18 User Defined Functions

3572 In computer programming, a **function** is a named section of code that can be  
3573 **called** from other sections of code. **Values** can be sent to a function for  
3574 processing as part of the **call** and a function always returns a value as its result.  
3575 A function can also generate side effects when it is called and side effects have  
3576 been covered in earlier sections.

3577 The values that are sent to a function when it is called are called **arguments** or  
3578 **actual parameters** and a function can accept 0 or more of them. These  
3579 arguments are placed within parentheses.

3580 MathPiper has many predefined functions (some of which have been discussed in  
3581 previous sections) but users can create their own functions too. The following  
3582 program creates a function called **addNums()** which takes two numbers as  
3583 arguments, adds them together, and returns their sum back to the calling code  
3584 as a result:

```
3585 In> addNums(num1,num2) := num1 + num2
3586 Result> True
```

3587 This line of code defined a new function called **addNums** and specified that it  
3588 will accept two values when it is called. The **first** value will be placed into the  
3589 variable **num1** and the **second** value will be placed into the variable **num2**.

3590 Variables like num1 and num2 which are used in a function to accept values from  
3591 calling code are called **formal parameters**. **Formal parameter variables** are  
3592 used inside a function to process the **values/actual parameters/arguments**  
3593 that were placed into them by the calling code.

3594 The code on the **right side** of the **assignment operator** is **bound** to the  
3595 function name "**addNums**" and it is executed each time **addNums()** is called.  
3596 The following example shows the new **addNums()** function being called multiple  
3597 times with different values being passed to it:

```
3598 In> addNums(2,3)
3599 Result> 5
```

```
3600 In> addNums(4,5)
3601 Result> 9
```

```
3602 In> addNums(9,1)
3603 Result> 10
```

3604 Notice that, unlike the functions that come with MathPiper, we chose to have this  
3605 function's name start with a **lower case letter**. We could have had addNums()  
3606 begin with an upper case letter but it is a **convention** in MathPiper for **user**

3607 **defined function names to begin with a lower case letter to distinguish**  
3608 **them from the functions that come with MathPiper.**

3609 The values that are returned from user defined functions can also be assigned to  
3610 variables. The following example uses a %mathpiper fold to define a function  
3611 called **evenIntegers()** and then this function is used in the MathPiper console:

```
3612 %mathpiper
3613 evenIntegers(endInteger) :=
3614 [
3615     resultList := {};
3616
3617     x := 2;
3618     While(x <= endInteger)
3619     [
3620         resultList := Append(resultList, x);
3621
3622         x := x + 2;
3623     ];
3624     /*
3625     The result of the last expression which is executed in a function
3626     is the result that the function returns to the caller. In this case,
3627     resultList is purposely being executed last so that its contents are
3628     returned to the caller.
3629     */
3630     resultList;
3631 ];
3632 %/mathpiper
3633
3634     %output,preserve="false"
3635     Result: True
3636 . %/output
3637
3638 In> a := evenIntegers(10)
3639 Result> {2,4,6,8,10}
3640
3641 In> Length(a)
3642 Result> 5
```

3640 The function **evenIntegers()** returns a list which contains all the even integers  
3641 from 2 up through the value that was passed into it. The fold was first executed  
3642 in order to define the **evenIntegers()** function and make it ready for use. The  
3643 **evenIntegers()** function was then called from the MathPiper console and 10  
3644 was passed to it.

3645 After the function was finished executing, it returned a list of even integers as a

3646 result and this result was assigned to the variable 'a'. We then passed the list  
3647 that was assigned to 'a' to the **Length()** function in order to determine its size.

## 3648 **18.1 Global Variables, Local Variables, & Local()**

3649 The new **evenIntegers()** function seems to work well, but there is a problem.  
3650 The variables 'x' and **resultList** were defined inside the function as **global**  
3651 **variables** which means they are accessible from anywhere, including from  
3652 within other functions, within other folds (as shown here):

```
3653 %mathpiper
3654 Echo(x, ",", resultList);
3655 %/mathpiper
3656     %output,preserve="false"
3657     Result: True
3658
3659     Side Effects:
3660     12 , {2,4,6,8,10}
3661 .    %/output
```

3662 and from within the MathPiper console:

```
3663 In> x
3664 Result> 12
3665 In> resultList
3666 Result> {2,4,6,8,10}
```

3667 **Using global variables inside of functions is usually not a good idea**  
3668 because code in other functions and folds might already be using (or will use) the  
3669 same variable names. Global variables which have the same name are the same  
3670 variable. When one section of code changes the value of a given global variable,  
3671 the value is changed everywhere that variable is used and this will eventually  
3672 cause problems.

3673 In order to prevent errors being caused by global variables having the same  
3674 name, a function named **Local()** can be called inside of a function to define what  
3675 are called **local variables**. A **local variable** is only accessible inside the  
3676 function it has been defined in, even if it has the same name as a global variable.  
3677 The following example shows a second version of the **evenIntegers()** function  
3678 which uses **Local()** to make 'x' and **resultList** local variables:

```
3679 %mathpiper
3680 /*
3681  This version of evenIntegers() uses Local() to make
3682  x and resultList local variables
3683  */
3684 evenIntegers(endInteger) :=
3685 [
3686     Local(x,resultList);
3687     resultList := {};
3688
3689     x := 2;
3690
3691     While(x <= endInteger)
3692     [
3693         resultList := Append(resultList, x);
3694
3695         x := x + 2;
3696     ];
3697     /*
3698     The result of the last expression which is executed in a function
3699     is the result that the function returns to the caller. In this case,
3700     resultList is purposely being executed last so that its contents are
3701     returned to the caller.
3702     */
3703     resultList;
3704 ];
3705 %/mathpiper
3706     %output,preserve="false"
3707     Result: True
3708 .    %/output
```

3709 We can verify that '**x**' and **resultList** are now local variables by first clearing  
3710 them, calling **evenIntegers()**, and then seeing what '**x**' and **resultList** contain:

```
3711 In> Clear(x, resultList)
3712 Result> True
3713 In> evenIntegers(10)
3714 Result> {2,4,6,8,10}
3715 In> x
3716 Result> x
3717 In> resultList
3718 Result> resultList
```

## 3719 18.2 Exercises

3720 For the following exercises, create a new MathRider worksheet file called  
3721 **book\_1\_section\_18\_exercises\_<your first name>\_<your last name>.mrw.**  
3722 **(Note: there are no spaces in this file name).** For example, John Smith's  
3723 worksheet would be called:

3724 **book\_1\_section\_18\_exercises\_john\_smith.mrw.**

3725 After this worksheet has been created, place your answer for each exercise that  
3726 requires a fold into its own fold in this worksheet. Place a title attribute in the  
3727 start tag of each fold which indicates the exercise the fold contains the solution  
3728 to. The folds you create should look similar to this one:

3729 `%mathpiper,title="Exercise 1"`

3730 `//Sample fold.`

3731 `%/mathpiper`

3732 If an exercise uses the MathPiper console instead of a fold, copy the work you  
3733 did in the console into the worksheet so it can be saved but do not put it in a fold.

### 3734 18.2.1 Exercise 1

3735 Carefully read all of section 18 up to this point. Evaluate each one of  
3736 the examples in the sections you read in the MathPiper worksheet you  
3737 created or in the MathPiper console and verify that the results match the  
3738 ones in the book. Copy all of the console examples you evaluated into your  
3739 worksheet so they will be saved but do not put them in a fold.

### 3740 18.2.2 Exercise 2

3741 Create a function called **tenOddIntegers()** which returns a list which  
3742 contains 10 random odd integers between 1 and 99 inclusive.

### 3743 18.2.3 Exercise 3

3744 Create a function called **convertStringToList(string)** which takes a string  
3745 as a parameter and returns a list which contains all of the characters in  
3746 the string. Here is an example of how the function should work:

3747 `In> convertStringToList("Hello friend!")`  
3748 `Result> {"H","e","l","l","o"," ","f","r","i","e","n","d","!"}`

3749 `In> convertStringToList("Computer Algebra System")`  
3750 `Result> {"C","o","m","p","u","t","e","r"," ","A","l","g","e","b","r","a"," "`  
3751 `","S","y","s","t","e","m"}`



## 3752 19 Miscellaneous topics

### 3753 19.1 Incrementing And Decrementing Variables With The ++ And -- 3754 Operators

3755 Up until this point we have been adding 1 to a variable with code in the form of **x**  
3756 **:= x + 1** and subtracting 1 from a variable with code in the form of **x := x - 1**.  
3757 Another name for **adding** 1 to a variable is **incrementing** it and **decrementing**  
3758 a variable means to **subtract** 1 from it. Now that you have had some experience  
3759 with these longer forms, it is time to show you shorter versions of them.

#### 3760 19.1.1 Incrementing Variables With The ++ Operator

3761 The number 1 can be added to a variable by simply placing the ++ operator after  
3762 it like this:

```
3763 In> x := 1  
3764 Result: 1
```

```
3765 In> x++;  
3766 Result: True
```

```
3767 In> x  
3768 Result: 2
```

3769 Here is a program that uses the ++ operator to increment a loop index variable:

```
3770 %mathpiper  
3771 count := 1;  
3772 While(count <= 10)  
3773 [  
3774     Echo(count);  
3775     count++; //The ++ operator increments the count variable.  
3776 ];  
3777  
3778 %/mathpiper  
3779 %output,preserve="false"  
3780 Result: True  
3781  
3782 Side Effects:  
3783 1  
3784 2
```

```
3785      3
3786      4
3787      5
3788      6
3789      7
3790      8
3791      9
3792     10
3793 .    %/output
```

## 3794 19.1.2 Decrementing Variables With The -- Operator

3795 The number 1 can be subtracted from a variable by simply placing the --  
3796 operator after it like this:

```
3797 In> x := 1
3798 Result: 1

3799 In> x--;
3800 Result: True

3801 In> x
3802 Result: 0
```

3803 Here is a program that uses the -- operator to decrement a loop index variable:

```
3804 %mathpiper

3805 count := 10;

3806 While(count >= 1)
3807 [
3808     Echo(count);
3809     count--; //The -- operator decrements the count variable.
3810 ]

3812 %/mathpiper

3813 %output,preserve="false"
3814 Result: True
3815
3816 Side Effects:
3817 10
3818 9
3819 8
3820 7
3821 6
3822 5
```

```
3823      4
3824      3
3825      2
3826      1
3827 .    %/output
```

## 3828 19.2 Exercises

3829 For the following exercises, create a new MathRider worksheet file called  
3830 **book\_1\_section\_19\_exercises\_<your first name>\_<your last name>.mrw.**  
3831 **(Note: there are no spaces in this file name).** For example, John Smith's  
3832 worksheet would be called:

3833 **book\_1\_section\_19\_exercises\_john\_smith.mrw.**

3834 After this worksheet has been created, place your answer for each exercise that  
3835 requires a fold into its own fold in this worksheet. Place a title attribute in the  
3836 start tag of each fold which indicates the exercise the fold contains the solution  
3837 to. The folds you create should look similar to this one:

```
3838 %mathpiper,title="Exercise 1"
```

```
3839 //Sample fold.
```

```
3840 %/mathpiper
```

3841 If an exercise uses the MathPiper console instead of a fold, copy the work you  
3842 did in the console into the worksheet so it can be saved but do not put it in a fold.

### 3843 19.2.1 Exercise 1

3844 Carefully read all of section 19 up to this point. Evaluate each one of  
3845 the examples in the sections you read in the MathPiper worksheet you  
3846 created or in the MathPiper console and verify that the results match the  
3847 ones in the book. Copy all of the console examples you evaluated into your  
3848 worksheet so they will be saved but do not put them in a fold.