

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.....	25
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.....	31
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 Exercises.....	39
8.3.1 Exercise 1.....	39
8.3.2 Exercise 2.....	39
8.3.3 Exercise 3.....	39
8.3.4 Exercise 4.....	39
9 MathPiper Programming Fundamentals.....	40

9.1 Values and Expressions.....	40
9.2 Operators.....	40
9.3 Operator Precedence.....	41
9.4 Changing The Order Of Operations In An Expression.....	42
9.5 Functions & Function Names.....	43
9.6 Functions That Produce Side Effects.....	44
9.6.1 Printing Related Functions: Echo(), Write(), And Newline().....	44
9.6.1.1 Echo().....	44
9.6.1.2 Echo Statements Are Useful For "Debugging" Programs.....	46
9.6.1.3 Write().....	47
9.6.1.4 NewLine().....	47
9.7 Expressions Are Separated By Semicolons.....	48
9.7.1 Placing More Than One Expression On A Line In A Fold.....	48
9.7.2 Placing More Than One Expression On A Line In The Console Using A Code Block.....	49
9.7.2.1 Automatic Bracket, Parentheses, And Brace Match Indicating....	50
9.8 Strings.....	51
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.....	52
9.8.2.1 Combining Strings With The : Operator.....	52
9.8.2.2 WriteString().....	53
9.8.2.3 Nl().....	53
9.8.2.4 Space().....	53
9.8.3 Accessing The Individual Letters In A String.....	53
9.9 Comments.....	54
9.10 Exercises.....	55
9.10.1 Exercise 1.....	56
9.10.2 Exercise 2.....	56
9.10.3 Exercise 3.....	56
9.10.4 Exercise 4.....	56
9.10.5 Exercise 5.....	56
9.10.6 Exercise 6.....	57
9.10.7 Exercise 7.....	57
10 Rectangular Selection Mode And Text Area Splitting.....	58
10.1 Rectangular Selection Mode.....	58
10.2 Text area splitting.....	58
10.3 Exercises.....	58
10.3.1 Exercise 1.....	59
11 Working With Random Integers.....	60
11.1 Obtaining Random Integers With The RandomInteger() Function.....	60

11.2 Simulating The Rolling Of Dice.....	61
11.3 Exercises.....	62
11.3.1 Exercise 1.....	62
12 Making Decisions.....	63
12.1 Conditional Operators.....	63
12.2 Predicate Expressions.....	66
12.3 Exercises.....	66
12.3.1 Exercise 1.....	66
12.3.2 Exercise 2.....	67
12.3.3 Exercise 3.....	67
12.4 Making Decisions With The If() Function & Predicate Expressions.....	67
12.4.1 If() Functions Which Include An "Else" Parameter.....	69
12.5 Exercises.....	69
12.5.1 Exercise 1.....	70
12.5.2 Exercise 2.....	70
12.6 The And(), Or(), & Not() Boolean Functions & Infix Notation.....	70
12.6.1 And().....	70
12.6.2 Or().....	72
12.6.3 Not() & Prefix Notation.....	73
12.7 Exercises.....	74
12.7.1 Exercise 1.....	75
12.7.2 Exercise 2.....	75
12.7.3 Exercise 3.....	75
13 The While() Looping Function & Bodied Notation.....	77
13.1 Printing The Integers From 1 to 10.....	77
13.2 Printing The Integers From 1 to 100.....	79
13.3 Printing The Odd Integers From 1 To 99.....	79
13.4 Printing The Integers From 1 To 100 In Reverse Order.....	80
13.5 Expressions Inside Of Code Blocks Are Indented.....	81
13.6 Long-Running Loops, Infinite Loops, & Interrupting Execution.....	81
13.7 A Program That Simulates Rolling Two Dice 50 Times.....	82
13.8 Exercises.....	84
13.8.1 Exercise 1.....	85
13.8.2 Exercise 2.....	85
13.8.3 Exercise 3.....	85
13.8.4 Exercise 4.....	85
14 Predicate Functions.....	86
14.1 Finding Prime Numbers With A Loop.....	87
14.2 Finding The Length Of A String With The Length() Function.....	89
14.3 Converting Numbers To Strings With The String() Function.....	90
14.4 Finding Prime Numbers Which End With 7 (And Multi-line Function Calls)	

.....	90
14.5 Exercises.....	91
14.5.1 Exercise 1.....	92
14.5.2 Exercise 2.....	92
14.5.3 Exercise 3.....	92
15 Lists: Values That Hold Sequences Of Expressions.....	93
15.1 Append() & Nondestructive List Operations.....	94
15.2 Using While Loops With Lists	95
15.2.1 Using A While Loop And Append() To Place Values In A List.....	96
15.3 Exercises.....	97
15.3.1 Exercise 1.....	98
15.3.2 Exercise 2.....	98
15.3.3 Exercise 3.....	98
15.3.4 Exercise 4.....	98
15.4 The ForEach() Looping Function.....	99
15.5 Print All The Values In A List Using A ForEach() function.....	99
15.6 Calculate The Sum Of The Numbers In A List Using ForEach().....	100
15.7 The .. Range Operator.....	101
15.8 Using ForEach() With The Range Operator To Print The Prime Numbers Between 1 And 100.....	101
15.8.1 Using ForEach() And The Range Operator To Place The Prime Numbers Between 1 And 50 Into A List.....	102
15.8.2 Exercises.....	103
15.8.3 Exercise 1.....	103
15.8.4 Exercise 2.....	104
15.8.5 Exercise 3.....	104
15.8.6 Exercise 4.....	104
15.8.7 Exercise 5.....	104
16 Functions & Operators Which Loop Internally.....	105
16.1 Functions & Operators Which Loop Internally To Process Lists.....	105
16.1.1 TableForm().....	105
16.1.2 Contains().....	105
16.1.3 Find().....	106
16.1.4 Count().....	106
16.1.5 Select().....	107
16.1.6 The Nth() Function & The [] Operator.....	107
16.1.7 The : Prepend Operator.....	108
16.1.8 Concat().....	108
16.1.9 Insert(), Delete(), & Replace().....	108
16.1.10 Take()	109
16.1.11 Drop().....	110

16.1.12 FillList()	110
16.1.13 RemoveDuplicates()	111
16.1.14 Reverse()	111
16.1.15 Partition()	111
16.1.16 Table()	112
16.1.17 HeapSort()	113
16.2 Functions That Work With Integers	113
16.2.1 RandomIntegerVector()	113
16.2.2 Max() & Min()	113
16.2.3 Div() & Mod()	114
16.2.4 Gcd()	115
16.2.5 Lcm()	115
16.2.6 Sum()	116
16.2.7 Product()	116
16.3 Exercises	116
16.3.1 Exercise 1	117
16.3.2 Exercise 2	117
16.3.3 Exercise 3	117
16.3.4 Exercise 4	117
16.3.5 Exercise 5	117
16.3.6 Exercise 6	117
17 Nested Loops	118
17.1 Generate All The Combinations That Can Be Entered Into A Two Digit Wheel Lock Using Two Nested Loops	118
17.2 Exercises	119
17.2.1 Exercise 1	120
17.2.2 Exercise 2	120
18 User Defined Functions	121
18.1 Global Variables, Local Variables, & Local()	123
18.2 Exercises	125
18.2.1 Exercise 1	125
18.2.2 Exercise 2	125
18.2.3 Exercise 3	125
19 Miscellaneous topics	126
19.1 Incrementing And Decrementing Variables With The ++ And -- Operators	126
19.1.1 Incrementing Variables With The ++ Operator	126
19.1.2 Decrementing Variables With The -- Operator	127
19.2 Exercises	128
19.2.1 Exercise 1	128

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

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

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:

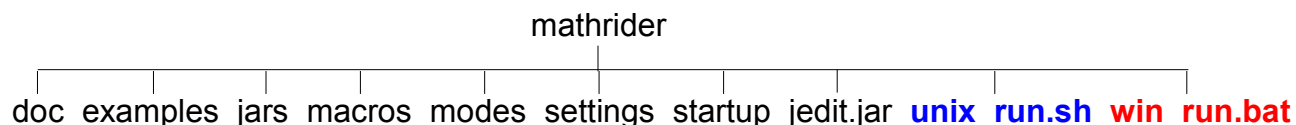


Illustration 1: MathRider's Directory Structure

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).

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 them.
427 The purpose of the **N()** function is to make sure that the information that is sent
428 to it is processed numerically instead of symbolically.

429 5.2.1.1 The Sqrt() Square Root Function

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

```
432 In> Sqrt(9)  
433 Result: 3
```

```
434 In> Sqrt(8)
435 Result: Sqrt(8)
```

```
436 In> N(Sqrt(8))
437 Result: 2.828427125
```

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

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

```
446 2.828427125
```

```
447 2.82842712474619
```

```
448 2.82842712474619009760337744842
```

```
449 2.8284271247461900976033774484193961571393437507539
```

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

454 5.2.1.2 The IsEven() Function

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

```
457 In> IsEven(4)
458 Result> True
```

```
459 In> IsEven(5)
460 Result> False
```

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

466 5.2.2 Accessing Previous Input And Results

467 The MathPiper console is like a mini text editor which means you can copy text

468 from it, paste text into it, and edit existing text. You can also reevaluate previous
469 input by simply placing the cursor on the desired **In>** line and pressing
470 **<shift><enter>** on it again.

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

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

```
477 In> 5*8
478 Result> 40
```

```
479 In> %
480 Result> 40
```

```
481 In> %*2
482 Result> 80
```

483 **5.3 Saving And Restoring A Console Session**

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

489 **5.3.1 Syntax Errors**

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

```
495 In> 12 \ 6
496 Error parsing expression, near token \
```

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

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

503 **5.4 Using The MathPiper Console As A Symbolic Calculator**

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

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

```
509 In> 1/2 + 1/3  
510 Result> 5/6
```

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

```
515 In> %  
516 Result> 5/6
```

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

```
519 In> Numerator(%)  
520 Result> 5
```

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

524 **5.4.1 Variables**

525 What would be nice is if MathPiper provided a way to store **results** (which are
526 also called **values**) in symbols that we choose instead of ones that it chooses.
527 Fortunately, this is exactly what it does! Symbols that can be associated with
528 values are called **variables**. Variable names must start with an upper or lower
529 case letter and be followed by zero or more upper case letters, lower case
530 letters, or numbers. Examples of variable names include: 'a', 'b', 'x', 'y', 'answer',
531 'totalAmount', and 'loop6'.

532 The process of associating a value with a variable is called **assigning** or **binding**
533 the value to the variable and this consists of placing the name of a **variable** you

would like to create on the **left** side of an assignment operator (**:=**) and an **expression** on the **right** side of this operator. When the expression returns a value, the value is assigned (or bound to) to the variable.

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

```
In> a := 1/2 + 1/3
Result> 5/6
```

```
In> a
Result> 5/6
```

```
In> Numerator(a)
Result> 5
```

```
In> Denominator(a)
Result> 6
```

In this example, the assignment operator (**:=**) was used to assign the result (or **value**) $\frac{5}{6}$ to the variable 'a'. **When 'a' was evaluated by itself, the value it was bound to (in this case $\frac{5}{6}$) was returned.** This value will stay bound to the variable 'a' as long as MathPiper is running unless 'a' is cleared with the **Clear()** function or 'a' has another value assigned to it. This is why we were able to determine both the numerator and the denominator of the rational number assigned to 'a' using two functions in turn.

5.4.1.1 Calculating With Unbound Variables

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

```
In> a := 9
Result> 9
```

```
In> a
Result> 9
```

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

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

```
In> a
Result> a
```

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

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

```
574 In> Factor(8)
575 Result> 2^3
```

```
576 In> Factor(14)
577 Result> 2*7
```

```
578 In> Factor(2343)
579 Result> 3*11*71
```

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

```
581 In> x
582 Result> x
```

```
583 In> IsBound(x)
584 Result> False
```

```
585 In> Factor(x^2 + 24*x + 80)
586 Result> (x+20)*(x+4)
```

```
587 In> Expand(%)
588 Result> x^2+24*x+80
```

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

592 What is more interesting, however, are the results returned by **Factor()** and
593 **Expand()**. **Factor()** is able to determine when expressions with unbound
594 variables are sent to it and it uses the rules of algebra to **manipulate** them into
595 factored form. The **Expand()** function was then able to take the factored
596 expression $(x+20)(x+4)$ and manipulate it until it was expanded. One way to
597 remember what the functions **Factor()** and **Expand()** do is to look at the second
598 letters of their names. The 'a' in **Factor** can be thought of as **adding**
599 parentheses to an expression and the 'x' in **Expand** can be thought of **xing** out
600 or removing parentheses from an expression.

601 **5.4.1.2 Variable And Function Names Are Case Sensitive**

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

```
608 In> Box := 1
609 Result> 1
```

```
610 In> box := 2
611 Result> 2
```

```
612 In> Box
613 Result> 1
```

```
614 In> box
615 Result> 2
```

616 **5.4.1.3 Using More Than One Variable**

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

```
619 a := 2
620 Result> 2
```

```
621 b := 3
622 Result> 3
```

```
623 a + b
624 Result> 5
```

```
625 answer := a + b
626 Result> 5
```

```
627 answer
628 Result> 5
```

629 The part of an expression that is on the **right side** of an assignment operator is
630 always evaluated first and the result is then assigned to the variable that is on
631 the **left side** of the operator.

632 Now that you have seen how to use the MathPiper console as both a **symbolic**

633 and a **numeric** calculator, our next step is to take a closer look at the functions
634 which are included with MathPiper. As you will soon discover, MathPiper
635 contains an amazing number of functions which deal with a wide range of
636 mathematics.

637 **5.5 Exercises**

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

640 **5.5.1 Exercise 1**

641 Carefully read all of section 5. Evaluate each one of the examples in
642 section 5 in the MathPiper console and verify that the results match the
643 ones in the book.

644 **5.5.2 Exercise 2**

645 Answer each one of the following questions:

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

647 b) What is a variable?

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

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

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

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

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

653 h) What does the % character do?

654 **5.5.3 Exercise 3**

655 Perform the following calculation:

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

656 **5.5.4 Exercise 4**

657 a) Assign the variable **answer** to the result of the calculation $\frac{1}{5} + \frac{7}{4} + \frac{15}{16}$
658 using the following line of code:

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

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

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

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

663 e) Use the Clear() function to unbind the variable **answer** and verify that

664 **answer** is unbound by executing the following code and by using the

665 IsBound() function:

666 In> **answer**

667 5.5.5 Exercise 5

668 Assign $\frac{1}{4}$ to variable **x**, $\frac{3}{8}$ to variable **y**, and $\frac{7}{16}$ to variable **z** using the

669 := operator. Then perform the following calculations:

670 a)

671 In> x

672 b)

673 In> y

674 c)

675 In> z

676 d)

677 In> x + y

678 d)

679 In> x + z

680 e)

681 In> x + y + z

682 **6 The MathPiper Documentation Plugin**

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

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

691 **6.1 Function List**

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

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

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

715 **6.2 Mini Web Browser Interface**

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

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

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

726 **6.3 Exercises**

727 **6.3.1 Exercise 1**

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

733 **6.3.2 Exercise 2**

734 Locate the `N()`, `IsEven()`, `IsOdd()`, `Clear()`, `IsBound()`, `Numerator()`,
735 `Denominator()`, and `Factor()` functions in the **User Functions** section of the
736 MathPiperDocs plugin and list which section each function is contained in.
737 Don't include the **Alphabetical** or **Built In** subsections in your search.

738 **7 Using MathRider As A Programmer's Text Editor**

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

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

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

752 **7.1 Creating, Opening, Saving, And Closing Text Files**

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

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

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

765 **7.2 Editing Files**

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

771 **7.3 File Modes**

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

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

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

781 ***7.4 Learning How To Type Properly Is An Excellent Investment Of Your*** 782 ***Time***

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

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

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

793 ***7.5 Exercises***

794 ***7.5.1 Exercise 1***

795 Carefully read all of section 7. Create a text file called
796 **"my_text_file.txt"** and place a few sentences in it. Save the text file
797 somewhere on your hard drive then close it. Now, open the text file again
798 using **File->Open** and verify that what you typed is still in the file.

799 8 MathRider Worksheet Files

800 While MathRider's ability to execute code inside a console provides a significant
801 amount of power to the user, most of MathRider's power is derived from
802 **worksheets**. MathRider worksheets are text files which have a **.mrw** extension
803 and are able to execute multiple types of code in a single text area. The
804 **worksheet_demo_1.mrw** file (which is preloaded in the MathRider environment
805 when it is first launched) demonstrates how a worksheet is able to execute
806 multiple types of code in what are called **code folds**.

807 8.1 Code Folds

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

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

```
818 %mathpiper
819 2 + 3;
820 %/mathpiper
```

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

```
825 %mathpiper
826 2 + 3;
827 %/mathpiper
828     %output,preserve="false"
829     Result: 5
830 .    %/output
```

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

832 %output have their **preserve property** set to **false**. This tells MathRider to
833 overwrite the %output fold with a new version during the next execution of its
834 parent. If preserve is set to **true**, the fold will not be overwritten and a new fold
835 will be created instead.

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

838 **8.1.1 The title Attribute**

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

```
842 %mathpiper,title="Add two numbers together."
```

```
843 2 + 3;
```

```
844 %/mathpiper
```

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

849 **8.2 Automatically Inserting Folds & Removing Unpreserved Folds**

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

```
851 %mathpiper
```

```
852 %/mathpiper
```

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

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

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

863 **8.3 Exercises**

864 A MathRider worksheet file called "**newbies_book_examples_1.mrw**" can be
865 obtained from this website:

866 [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)
867 [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)

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

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

875 **8.3.1 Exercise 1**

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

879 **8.3.2 Exercise 2**

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

882 **8.3.3 Exercise 3**

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

884 **8.3.4 Exercise 4**

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

888 a) $2 * 7 + 3$

889 b) $18 / 3$

890 c) $234238342 + 2038408203$

891 d) $324802984 * 2308098234$

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

893 9 MathPiper Programming Fundamentals

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

898 9.1 Values and Expressions

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

901 3

902 represents the number three, the value:

903 0.5

904 represents the number one half, and the value:

905 "Mathematics is powerful!"

906 represents an English sentence.

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

910 3

911 2 + 3

912 5 + 6*21/18 - 2^3

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

916 In> 2 + 3

917 Result> 5

918 9.2 Operators

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

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

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

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

929 In> 5 - 2
930 Result> 3

931 In> 3*4
932 Result> 12

933 In> 30/3
934 Result> 10

935 In> 8%5
936 Result> 3

937 In> 2^3
938 Result> 8

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

940 In> -3
941 Result> -3

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

944 In> - -3
945 Result> 3

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

951 **9.3 Operator Precedence**

952 When expressions contain more than one operator, MathPiper uses a set of rules
953 called **operator precedence** to determine the order in which the operators are
954 applied to the values in the expression. Operator precedence is also referred to
955 as the **order of operations**. Operators with higher precedence are evaluated
956 before operators with lower precedence. The following table shows a subset of
957 MathPiper's operator precedence rules with higher precedence operators being
958 placed higher in the table:

959 [^] Exponents are evaluated right to left.

960 *,%,/ Then multiplication, remainder, and division operations are evaluated

961 left to right.

962 +, - Finally, addition and subtraction are evaluated left to right.

963 Lets manually apply these precedence rules to the multi-operator expression we

964 used earlier. Here is the expression in source code form:

965 5 + 6*21/18 - 2^3

966 And here it is in traditional form:

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

967 According to the precedence rules, this is the order in which MathPiper

968 evaluates the operations in this expression:

969 5 + 6*21/18 - 2^3

970 5 + 6*21/18 - 8

971 5 + 126/18 - 8

972 5 + 7 - 8

973 12 - 8

974 4

975 Starting with the first expression, MathPiper evaluates the [^] operator first which

976 results in the 8 in the expression below it. In the second expression, the *

977 operator is executed next, and so on. The last expression shows that the final

978 result after all of the operators have been evaluated is 4.

979 **9.4 Changing The Order Of Operations In An Expression**

980 The default order of operations for an expression can be changed by grouping

981 various parts of the expression within parentheses (). Parentheses force the

982 code that is placed inside of them to be evaluated before any other operators are

983 evaluated. For example, the expression 2 + 4*5 evaluates to 22 using the

984 default precedence rules:

985 In> 2 + 4*5

986 Result> 22

987 If parentheses are placed around 4 + 5, however, the addition operator is forced

988 to be evaluated before the multiplication operator and the result is 30:

```
989 In> (2 + 4)*5
990 Result> 30
```

991 Parentheses can also be nested and nested parentheses are evaluated from the
992 most deeply nested parentheses outward:

```
993 In> ((2 + 4)*3)*5
994 Result> 90
```

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

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

- 999 () Parentheses are evaluated from the inside out.
- 1000 ^ Then exponents are evaluated right to left.
- 1001 *,%/, Then multiplication, remainder, and division operations are evaluated
1002 left to right.
- 1003 +, - Finally, addition and subtraction are evaluated left to right.

1004 **9.5 Functions & Function Names**

1005 In programming, **functions** are named blocks of code that can be executed one
1006 or more times by being **called** from other parts of the same program or called
1007 from other programs. Functions **can have values passed to them** from the
1008 calling code and they **always return a value** back to the calling code when they
1009 are finished executing. An example of a function is the **IsEven()** function which
1010 was discussed in an previous section.

1011 Functions are one way that MathPiper enables code to be reused. Most
1012 programming languages allow code to be reused in this way, although in other
1013 languages these named blocks of code are sometimes called **subroutines**,
1014 **procedures**, or **methods**.

1015 The functions that come with MathPiper have names which consist of either a
1016 single word (such as **Sum()**) or multiple words that have been put together to
1017 form a compound word (such as **IsBound()**). All letters in the names of
1018 functions which come with MathPiper are lower case except the beginning letter
1019 in each word, which are upper case.

1020 **9.6 Functions That Produce Side Effects**

1021 Most functions are executed to obtain the **results** they produce but some
1022 functions are executed in order to **have them perform work that is not in the**
1023 **form of a result**. Functions that perform work that is not in the form of a result
1024 are said to produce **side effects**. Side effects include many forms of work such
1025 as sending information to the user, opening files, and changing values in the
1026 computer's memory.

1027 When a function produces a side effect which sends information to the user, this
1028 information has the words **Side Effects:** placed before it in the output instead of
1029 the word **Result:**. The **Echo()** and **Write()** functions are examples of functions
1030 that produce side effects and they are covered in the next section.

1031 **9.6.1 Printing Related Functions: Echo(), Write(), And Newline()**

1032 The printing related functions send text information to the user and this is
1033 usually referred to as "printing" in this document. However, it may also be called
1034 "echoing" and "writing".

1035 **9.6.1.1 Echo()**

1036 The **Echo()** function takes one expression (or multiple expressions separated by
1037 commas) evaluates each expression, and then prints the results as side effect
1038 output. The following examples illustrate this:

```
1039 In> Echo(1)
1040 Result> True
1041 Side Effects>
1042 1
```

1043 In this example, the number 1 was passed to the Echo() function, the number
1044 was evaluated (all numbers evaluate to themselves), and the result of the
1045 evaluation was then printed as a side effect. Notice that Echo() **also returned a**
1046 **result**. In MathPiper, all functions return a result, but functions whose main
1047 purpose is to produce a side effect usually just return a result of **True** if the side
1048 effect succeeded or **False** if it failed. In this case, Echo() returned a result of
1049 **True** because it was able to successfully print a 1 as its side effect.

1050 The next example shows multiple expressions being sent to Echo() (notice that
1051 the expressions are separated by commas):

```
1052 In> Echo(1,1+2,2*3)
1053 Result> True
1054 Side Effects>
1055 1 3 6
```

1056 The expressions were each evaluated and their results were returned (separated
1057 by spaces) as side effect output. If it is desired that commas be printed between
1058 the numbers in the output, simply place three commas between the expressions
1059 that are passed to Echo():

```
1060 In> Echo(1,,,1+2,,,2*3)
1061 Result> True
1062 Side Effects>
1063 1 , 3 , 6
```

1064 Each time an Echo() function is executed, it always forces the display to drop
1065 down to the next line after it is finished. This can be seen in the following
1066 program which is similar to the previous one except it uses a separate Echo()
1067 function to display each expression:

```
1068 %mathpiper
1069 Echo(1);
1070 Echo(1+2);
1071 Echo(2*3);
1072 %/mathpiper
1073 %output,preserve="false"
1074 Result: True
1075
1076 Side Effects:
1077 1
1078 3
1079 6
1080 . %/output
```

1081 Notice how the 1, the 3, and the 6 are each on their own line.

1082 Now that we have seen how Echo() works, lets use it to do something useful. If
1083 more than one expression is evaluated in a %mathpiper fold, only the result from
1084 the last expression that was evaluated (which is usually the bottommost
1085 expression) is displayed:

```
1086 %mathpiper
1087 a := 1;
1088 b := 2;
1089 c := 3;
1090 %/mathpiper
```

```
1091     %output,preserve="false"
1092     Result: 3
1093 .    %/output
```

1094 In MathPiper, programs are executed one line at a time, starting at the topmost
1095 line of code and working downwards from there. In this example, the line `a := 1;`
1096 is executed first, then the line `b := 2;` is executed, and so on. Notice, however,
1097 that even though we wanted to see what was in all three variables, only the
1098 content of the last variable was displayed.

1099 The following example shows how `Echo()` can be used to display the contents of
1100 all three variables:

```
1101 %mathpiper
1102 a := 1;
1103 Echo(a);
1104 b := 2;
1105 Echo(b);
1106 c := 3;
1107 Echo(c);
1108 %/mathpiper
1109     %output,preserve="false"
1110     Result: True
1111
1112     Side Effects:
1113     1
1114     2
1115     3
1116 .    %/output
```

1117 9.6.1.2 Echo Statements Are Useful For "Debugging" Programs

1118 The errors that are in a program are often called "bugs". This name came from
1119 the days when computers were the size of large rooms and were made using
1120 electromechanical parts. Periodically, bugs would crawl into the machines and
1121 interfere with its moving mechanical parts and this would cause the machine to
1122 malfunction. The bugs needed to be located and removed before the machine
1123 would run properly again.

1124 Of course, even back then most program errors were produced by programmers
1125 entering wrong programs or entering programs wrong, but they liked to say that
1126 all of the errors were caused by bugs and not by themselves! The process of
1127 fixing errors in a program became known as **debugging** and the names "bugs"

1128 and "debugging" are still used by programmers today.

1129 One of the standard ways to locate bugs in a program is to place **Echo()** function
1130 calls in the code at strategic places which **print the contents of variables and**
1131 **display messages**. These Echo() functions will enable you to see what your
1132 program is doing while it is running. After you have found and fixed the bugs in
1133 your program, you can remove the debugging Echo() function calls or comment
1134 them out if you think they may be needed later.

1135 **9.6.1.3 Write()**

1136 The **Write()** function is similar to the Echo() function except it does not
1137 automatically drop the display down to the next line after it finishes executing:

```
1138 %mathpiper
1139 Write(1);
1140 Write(1+2);
1141 Echo(2*3);
1142 %/mathpiper
1143     %output,preserve="false"
1144     Result: True
1145
1146     Side Effects:
1147     1 3 6
1148 .    %/output
```

1149 Write() and Echo() have other differences besides the one discussed here and
1150 more information about them can be found in the documentation for these
1151 functions.

1152 **9.6.1.4 NewLine()**

1153 The **NewLine()** function simply prints a blank line in the side effects output. It
1154 is useful for placing vertical space between printed lines:

```
1155 %mathpiper
1156 a := 1;
1157 Echo(a);
1158 NewLine();
1159 b := 2;
1160 Echo(b);
```

```
1161 NewLine();
1162 c := 3;
1163 Echo(c);

1164 %mathpiper

1165     %output,preserve="false"
1166     Result: True
1167
1168     Side Effects:
1169     1
1170
1171     2
1172
1173     3
1174 . %/output
```

1173 9.7 Expressions Are Separated By Semicolons

1174 As discussed earlier, all of the expressions that are inside of a **%mathpiper** fold
1175 must have a semicolon (;) after them. However, the expressions executed in the
1176 **MathPiper console** did not have a semicolon after them. MathPiper actually
1177 requires that all expressions end with a semicolon, but one does not need to add
1178 a semicolon to an expression which is typed into the MathPiper console **because**
1179 **the console adds it automatically** when the expression is executed.

1180 9.7.1 Placing More Than One Expression On A Line In A Fold

1181 All the previous code examples have had each of their expressions on a separate
1182 line, but multiple expressions can also be placed on a single line because the
1183 semicolons tell MathPiper where one expression ends and the next one begins:

```
1184 %mathpiper

1185 a := 1; Echo(a); b := 2; Echo(b); c := 3; Echo(c);

1186 %/mathpiper

1187     %output,preserve="false"
1188     Result: True
1189
1190     Side Effects:
1191     1
1192     2
1193     3
1194 . %/output
```

1195 The spaces that are in the code of this example are used to make the code more
1196 readable. Any spaces that are present within any expressions or between them
1197 are ignored by MathPiper and if we remove the spaces from the previous code,
1198 the output remains the same:

```
1199 %mathpiper
1200 a:=1;Echo(a);b:=2;Echo(b);c:= 3;Echo(c);
1201 %/mathpiper
1202     %output,preserve="false"
1203     Result: True
1204
1205     Side Effects:
1206     1
1207     2
1208     3
1209 .    %/output
```

1210 9.7.2 Placing More Than One Expression On A Line In The Console Using 1211 A Code Block

1212 The MathPiper console is only able to execute one expression at a time so if the
1213 previous code that executes three variable assignments and three Echo()
1214 functions on a single line is evaluated in the console, only the expression **a := 1**
1215 is executed:

```
1216 In> a := 1; Echo(a); b := 2; Echo(b); c := 3; Echo(c);
1217 Result> 1
```

1218 Fortunately, this limitation can be overcome by placing the code into a **code**
1219 **block**. A **code block** (which is also called a **compound expression**) consists of
1220 one or more expressions which are separated by semicolons and placed within an
1221 open bracket ([) and close bracket (]) pair. If a code block is evaluated in the
1222 MathPiper console, each expression in the block will be executed from left to
1223 right. The following example shows the previous code being executed within of a
1224 code block inside the MathPiper console:

```
1225 In> [a := 1; Echo(a); b := 2; Echo(b); c := 3; Echo(c);]
1226 Result> True
1227 Side Effects>
1228 1
1229 2
1230 3
```

1231 Notice that this time all of the expressions were executed and 1-3 was printed as

1232 a side effect. Code blocks always return the result of the last expression
1233 executed as the result of the whole block. In this case, True was returned as the
1234 result because the last Echo(c) function returned True. If we place another
1235 expression after the Echo(c) function, however, the block will execute this new
1236 expression last and its result will be the one returned by the block:

```
1237 In> [a := 1; Echo(a); b := 2; Echo(b); c := 3; Echo(c); 2 + 2]
1238 Result> 4
1239 Side Effects>
1240 1
1241 2
1242 3
```

1243 Finally, code blocks can have their contents placed on separate lines if desired:

```
1244 %mathpiper
1245 [
1246     a := 1;
1247     Echo(a);
1248     b := 2;
1249     Echo(b);
1250     c := 3;
1251     Echo(c);
1252 ];
1253
1254 %/mathpiper
1255
1256 %output,preserve="false"
1257 Result: True
1258
1259 Side Effects:
1260 1
1261 2
1262 3
1263 . %/output
```

1267 Code blocks are very powerful and we will be discussing them further in later
1268 sections.

1269 9.7.2.1 Automatic Bracket, Parentheses, And Brace Match Indicating

1270 In programming, most open brackets '[' have a close bracket ']', most open
1271 parentheses '(' have a close parentheses ')', and most open braces '{' have a
1272 close brace '}'. It is often difficult to make sure that each "open" character has a

1273 matching "close" character and if any of these characters don't have a match,
1274 then an error will be produced.

1275 Thankfully, most programming text editors have a character match indicating
1276 tool that will help locate problems. To try this tool, paste the following code into
1277 a .mrw file and following the directions that are present in its comments:

```
1278 %mathpiper
1279 /*
1280     Copy this code into a .mrw file. Then, place the cursor
1281     to the immediate right of any {, }, [, ], (, or ) character.
1282     You should notice that the match to this character is
1283     indicated by a rectangle being drawing around it.
1284 */
```

```
1285 list := {1,2,3};
1286 [
1287     Echo("Hello");
1288     Echo(list);
1289 ];
```

```
1290 %/mathpiper
```

1291 9.8 Strings

1292 A **string** is a **value** that is used to hold text-based information. The typical
1293 expression that is used to create a string consists of **text which is enclosed**
1294 **within double quotes**. Strings can be assigned to variables just like numbers
1295 can and strings can also be displayed using the Echo() function. The following
1296 program assigns a string value to the variable 'a' and then echos it to the user:

```
1297 %mathpiper
1298 a := "Hello, I am a string.";
1299 Echo(a);
1300 %/mathpiper
1301 %output,preserve="false"
1302 Result: True
1303
1304 Side Effects:
1305 Hello, I am a string.
1306 . %/output
```

1307 9.8.1 The MathPiper Console and MathPiper Folds Can Access The Same 1308 Variables

1309 A useful aspect of using MathPiper inside of MathRider is that variables that are
1310 assigned inside of a **%mathpiper fold** are accessible inside of the **MathPiper**
1311 **console** and variables that are assigned inside of the **MathPiper console** are
1312 available inside of **%mathpiper folds**. For example, after the above fold is
1313 executed, the string that has been bound to variable 'a' can be displayed in the
1314 MathPiper console:

```
1315 In> a  
1316 Result> "Hello, I am a string."
```

1317 9.8.2 Using Strings To Make Echo's Output Easier To Read

1318 When the Echo() function is used to print the values of multiple variables, it is
1319 often helpful to print some information next to each variable so that it is easier to
1320 determine which value came from which Echo() function in the code. The
1321 following program prints the name of the variable that each value came from
1322 next to it in the side effects output:

```
1323 %mathpiper  
1324 a := 1;  
1325 Echo("Variable a: ", a);  
  
1326 b := 2;  
1327 Echo("Variable b: ", b);  
  
1328 c := 3;  
1329 Echo("Variable c: ", c);  
  
1330 %/mathpiper  
  
1331 %output,preserve="false"  
1332 Result: True  
1333  
1334 Side Effects:  
1335 Variable a: 1  
1336 Variable b: 2  
1337 Variable c: 3  
1338 . %/output
```

1339 9.8.2.1 Combining Strings With The : Operator

1340 If you need to combine two or more strings into one string, you can use the :
1341 operator like this:

```
1342 In> "A" : "B" : "C"
1343 Result: "ABC"
```

```
1344 In> "Hello " : "there!"
1345 Result: "Hello there!"
```

1346 **9.8.2.2 WriteString()**

1347 The **WriteString()** function prints a string without shows the double quotes that
1348 are around it.. For example, here is the Write() function being used to print the
1349 string "Hello":

```
1350 In> Write("Hello")
1351 Result: True
1352 Side Effects:
1353 "Hello"
```

1354 Notice the double quotes? Here is how the WriteString() function prints "Hello":

```
1355 In> WriteString("Hello")
1356 Result: True
1357 Side Effects:
1358 Hello
```

1359 **9.8.2.3 NI()**

1360 The **NI()** (New Line) function is used with the : function to place newline
1361 characters inside of strings:

```
1362 In> WriteString("A": NI() : "B")
1363 Result: True
1364 Side Effects:
1365 A
1366 B
```

1367 **9.8.2.4 Space()**

1368 The Space() function is used to add spaces to printed output:

```
1369 In> WriteString("A"); Space(10); WriteString("B")
1370 Result: True
1371 Side Effects:
1372 A          B
```

1373 **9.8.3 Accessing The Individual Letters In A String**

1374 Individual letters in a string (which are also called **characters**) can be accessed

1375 by placing the character's position number (also called an **index**) inside of
1376 brackets **[]** after the variable it is bound to. A character's position is determined
1377 by its distance from the left side of the string starting at 1. For example, in the
1378 string "Hello", 'H' is at position 1, 'e' is at position 2, etc. The following code
1379 shows individual characters in the above string being accessed:

```
1380 In> a := "Hello, I am a string."  
1381 Result> "Hello, I am a string."
```

```
1382 In> a[1]  
1383 Result> "H"
```

```
1384 In> a[2]  
1385 Result> "e"
```

```
1386 In> a[3]  
1387 Result> "l"
```

```
1388 In> a[4]  
1389 Result> "l"
```

```
1390 In> a[5]  
1391 Result> "o"
```

1392 9.9 Comments

1393 Source code can often be difficult to understand and therefore all programming
1394 languages provide the ability for **comments** to be included in the code.

1395 Comments are used to explain what the code near them is doing and they are
1396 usually meant to be read by humans instead of being processed by a computer.
1397 Therefore, comments are ignored by the computer when a program is executed.

1398 There are two ways that MathPiper allows comments to be added to source code.
1399 The first way is by placing two forward slashes **//** to the left of any text that is
1400 meant to serve as a comment. The text from the slashes to the end of the line
1401 the slashes are on will be treated as a comment. Here is a program that contains
1402 comments which use slashes:

```
1403 %mathpiper  
1404 //This is a comment.
```

```
1405 x := 2; //Set the variable x equal to 2.
```

```
1406 %/mathpiper
```

```
1407     %output,preserve="false"  
1408     Result: 2  
1409 .    %/output
```

1410 When this program is executed, any text that starts with slashes is ignored.

1411 The second way to add comments to a MathPiper program is by enclosing the
1412 comments inside of slash-asterisk/asterisk-slash symbols `/* */`. This option is
1413 useful when a comment is too large to fit on one line. Any text between these
1414 symbols is ignored by the computer. This program shows a longer comment
1415 which has been placed between these symbols:

```
1416 %mathpiper
1417 /*
1418  This is a longer comment and it uses
1419  more than one line. The following
1420  code assigns the number 3 to variable
1421  x and then returns it as a result.
1422  */
1423 x := 3;
1424 %/mathpiper
1425     %output,preserve="false"
1426     Result: 3
1427 .    %/output
```

1428 9.10 Exercises

1429 For the following exercises, create a new MathRider worksheet file called
1430 **book_1_section_9_exercises_<your first name>_<your last name>.mrw**.
1431 (**Note: there are no spaces in this file name**). For example, John Smith's
1432 worksheet would be called:

1433 **book_1_section_9_exercises_john_smith.mrw**.

1434 After this worksheet has been created, place your answer for each exercise that
1435 requires a fold into its own fold in this worksheet. Place a title attribute in the
1436 start tag of each fold which indicates the exercise the fold contains the solution
1437 to. The folds you create should look similar to this one:

```
1438 %mathpiper,title="Exercise 1"
1439 //Sample fold.
1440 %/mathpiper
```

1441 If an exercise uses the MathPiper console instead of a fold, copy the work you
1442 did in the console into the worksheet so it can be saved.

1443 9.10.1 Exercise 1

1444 Carefully read all of section 9. Evaluate each one of the examples in
1445 section 9 in the MathPiper worksheet you created or in the MathPiper
1446 console and verify that the results match the ones in the book. Copy all
1447 of the console examples you evaluated into your worksheet so they will be
1448 saved.

1449 9.10.2 Exercise 2

1450 Change the precedence of the following expression using parentheses so that
1451 it prints 20 instead of 14:

1452 $2 + 3 * 4$

1453 9.10.3 Exercise 3

1454 Place the following calculations into a fold and then use one Echo()
1455 function per variable to print the results of the calculations. Put
1456 strings in the Echo() functions which indicate which variable each
1457 calculated value is bound to:

1458 $a := 1+2+3+4+5;$
1459 $b := 1-2-3-4-5;$
1460 $c := 1*2*3*4*5;$
1461 $d := 1/2/3/4/5;$

1462 9.10.4 Exercise 4

1463 Place the following calculations into a fold and then use one Echo()
1464 function to print the results of all the calculations on a single line
1465 (Remember, the Echo() function can print multiple values if they are
1466 separated by commas.):

1467 `Clear(x);`
1468 $a := 2*2*2*2*2;$
1469 $b := 2^5;$
1470 $c := x^2 * x^3;$
1471 $d := 2^2 * 2^3;$

1472 9.10.5 Exercise 5

1473 The following code assigns a string which contains all of the upper case
1474 letters of the alphabet to the variable **upper**. Each of the three Echo()
1475 functions prints an index number and the letter that is at that position in
1476 the string. Place this code into a fold and then continue the Echo()
1477 functions so that all 26 letters and their index numbers are printed

1478 `upper := "ABCDEFGHIJKLMNOPQRSTUVWXYZ";`

1479 `Echo(1,upper[1]);`
1480 `Echo(2,upper[2]);`

```
1481 Echo(3,upper[3]);
```

1482 9.10.6 Exercise 6

1483 Use Echo() functions to print an index number and the character at this
1484 position for the following string (this is similar to what was done in
1485 Exercise 4.):

```
1486 extra := "!.@#$$%^&*() _+<>,?/{ }[]|\-= ";
```

```
1487 Echo(1,extra[1]);
```

```
1488 Echo(2,extra[2]);
```

```
1489 Echo(3,extra[3]);
```

1490 9.10.7 Exercise 7

1491 The following program uses strings and index numbers to print a person's
1492 name. Create a program which uses the three strings from this program to
1493 print the names of three of your favorite movie actors.

```
1494 %mathpiper
```

```
1495 /*
```

```
1496     This program uses strings and index numbers to print
```

```
1497     a person's name.
```

```
1498 */
```

```
1499 upper := "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

```
1500 lower := "abcdefghijklmnopqrstuvwxyz";
```

```
1501 extra := "!.@#$$%^&*() _+<>,?/{ }[]|\-= ";
```

```
1502 //Print "Mary Smith.".
```

```
1503 Echo(upper[13],lower[1],lower[18],lower[25],extra[12],upper[19],lower[13],l
```

```
1504 ower[9],lower[20],lower[8],extra[1]);
```

```
1505 %/mathpiper
```

```
1506     %output,preserve="false"
```

```
1507     Result: True
```

```
1508
```

```
1509     Side Effects:
```

```
1510     Mary Smith.
```

```
1511 .    %/output
```


1512 10 Rectangular Selection Mode And Text Area Splitting

1513 10.1 Rectangular Selection Mode

1514 One capability that MathRider has that a word processor may not have is the
1515 ability to select rectangular sections of text. To see how this works, do the
1516 following:

- 1517 1) Type three or four lines of text into a text area.
- 1518 2) Hold down the **<Alt>** key then slowly press the **backslash key** (\) a few
1519 times. The bottom of the MathRider window contains a text field which
1520 MathRider uses to communicate information to the user. As **<Alt>** is
1521 repeatedly pressed, messages are displayed which read **Rectangular**
1522 **selection is on** and **Rectangular selection is off**.
- 1523 3) Turn rectangular selection on and then select some text in order to see
1524 how this is different than normal selection mode. **When you are done**
1525 **experimenting, set rectangular selection mode to off.**

1526 Most of the time normal selection mode is what you want to use but in certain
1527 situations rectangular selection mode is better.

1528 10.2 Text area splitting

1529 Sometimes it is useful to have two or more text areas open for a single document
1530 or multiple documents so that different parts of the documents can be edited at
1531 the same time. A situation where this would have been helpful was in the
1532 previous section where the output from an exercise in a MathRider worksheet
1533 contained a list of index numbers and letters which was useful for completing a
1534 later exercise.

1535 MathRider has this ability and it is called **splitting**. If you look just to the right
1536 of the toolbar there is an icon which looks like a blank window, an icon to the
1537 right of it which looks like a window which was split horizontally, and an icon to
1538 the right of the horizontal one which is split vertically. If you let your mouse
1539 hover over these icons, a short description will be displayed for each of them.

1540 Select a text area and then experiment with splitting it by pressing the horizontal
1541 and vertical splitting buttons. Move around these split text areas with their
1542 scroll bars and when you want to unsplit the document, just press the "**Unsplit**
1543 **All**" icon.

1544 10.3 Exercises

1545 For the following exercises, create a new MathRider worksheet file called
1546 **book_1_section_10_exercises_<your first name>_<your last name>.mrw**.

1547 (**Note: there are no spaces in this file name**). For example, John Smith's
1548 worksheet would be called:

1549 **book_1_section_10_exercises_john_smith.mrw.**

1550 For the following exercises, simply type your answers anywhere in the
1551 worksheet.

1552 **10.3.1 Exercise 1**

1553 Carefully read all of section 9 then answer the following questions:

1554 a) Give two examples where rectangular selection mode may be more useful
1555 than regular selection mode.

1556 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
```

```
1597 In> RandomInteger(100)
1598 Result> 82
1599 In> RandomInteger(100)
1600 Result> 93
1601 In> RandomInteger(100)
1602 Result> 32
```

1603 A range of random integers that does not start with 1 can also be generated by
1604 using the **two argument** version of **RandomInteger()**. For example, random
1605 integers between 25 and 75 can be obtained by passing RandomInteger() the
1606 lowest integer in the range and the highest one:

```
1607 In> RandomInteger(25, 75)
1608 Result: 28
1609 In> RandomInteger(25, 75)
1610 Result: 37
1611 In> RandomInteger(25, 75)
1612 Result: 58
1613 In> RandomInteger(25, 75)
1614 Result: 50
1615 In> RandomInteger(25, 75)
1616 Result: 70
```

1617 **11.2 Simulating The Rolling Of Dice**

1618 The following example shows the simulated rolling of a single six sided die using
1619 the RandomInteger() function:

```
1620 In> RandomInteger(6)
1621 Result> 5
1622 In> RandomInteger(6)
1623 Result> 6
1624 In> RandomInteger(6)
1625 Result> 3
1626 In> RandomInteger(6)
1627 Result> 2
1628 In> RandomInteger(6)
1629 Result> 5
```

1630 Code that simulates the rolling of two 6 sided dice can be evaluated in the
1631 MathPiper console by placing it within a **code block**. The following code
1632 outputs the sum of the two simulated dice:

```
1633 In> [a := RandomInteger(6); b := RandomInteger(6); a + b;]
1634 Result> 6
1635 In> [a := RandomInteger(6); b := RandomInteger(6); a + b;]
1636 Result> 12
1637 In> [a := RandomInteger(6); b := RandomInteger(6); a + b;]
1638 Result> 6
```

```
1639 In> [a := RandomInteger(6); b := RandomInteger(6); a + b;]
1640 Result> 4
1641 In> [a := RandomInteger(6); b := RandomInteger(6); a + b;]
1642 Result> 3
1643 In> [a := RandomInteger(6); b := RandomInteger(6); a + b;]
1644 Result> 8
```

1645 Now that we have the ability to simulate the rolling of two 6 sided dice, it would
1646 be interesting to determine if some sums of these dice occur more frequently
1647 than other sums. What we would like to do is to roll these simulated dice
1648 hundreds (or even thousands) of times and then analyze the sums that were
1649 produced. We don't have the programming capability to easily do this yet, but
1650 after we finish the section on **while loops**, we will.

1651 11.3 Exercises

1652 For the following exercises, create a new MathRider worksheet file called
1653 **book_1_section_11_exercises_<your first name>_<your last name>.mrw.**
1654 (**Note: there are no spaces in this file name**). For example, John Smith's
1655 worksheet would be called:

1656 **book_1_section_11_exercises_john_smith.mrw.**

1657 After this worksheet has been created, place your answer for each exercise that
1658 requires a fold into its own fold in this worksheet. Place a title attribute in the
1659 start tag of each fold which indicates the exercise the fold contains the solution
1660 to. The folds you create should look similar to this one:

```
1661 %mathpiper,title="Exercise 1"
1662 //Sample fold.
1663 %/mathpiper
```

1664 If an exercise uses the MathPiper console instead of a fold, copy the work you
1665 did in the console into the worksheet so it can be saved.

1666 11.3.1 Exercise 1

1667 Carefully read all of section 11. Evaluate each one of the examples in
1668 section 11 in the MathPiper worksheet you created or in the MathPiper
1669 console and verify that the results match the ones in the book. Copy all
1670 of the console examples you evaluated into your worksheet so they will be
1671 saved.

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 True if the two values are equal and False 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 True if the values are not equal and False if they are equal.
<code>x < y</code>	Returns True if the left value is less than the right value and False if the left value is not less than the right value.
<code>x <= y</code>	Returns True if the left value is less than or equal to the right value and False if the left value is not less than or equal to the right value.
<code>x > y</code>	Returns True if the left value is greater than the right value and False if the left value is not greater than the right value.
<code>x >= y</code>	Returns True if the left value is greater than or equal to the right value and False 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
```

```
1692 In> 4 > 5
1693 Result> False
```

```
1694 In> 8 >= 8
1695 Result> True
```

```
1696 In> 5 <= 10
1697 Result> True
```

1698 The following examples show each of the conditional operators in Table 2 being
1699 used to compare values that have been assigned to variables **x** and **y**:

```
1700 %mathpiper
```

```
1701 // Example 1.
1702 x := 2;
1703 y := 3;
```

```
1704 Echo(x, "=", y, ":", x = y);
1705 Echo(x, "!= ", y, ":", x != y);
1706 Echo(x, "< ", y, ":", x < y);
1707 Echo(x, "<= ", y, ":", x <= y);
1708 Echo(x, "> ", y, ":", x > y);
1709 Echo(x, ">= ", y, ":", x >= y);
```

```
1710 %/mathpiper
```

```
1711 %output,preserve="false"
1712 Result: True
1713
1714 Side Effects:
1715 2 = 3 :False
1716 2 != 3 :True
1717 2 < 3 :True
1718 2 <= 3 :True
1719 2 > 3 :False
1720 2 >= 3 :False
1721 . %/output
```

```
1722 %mathpiper
```

```
1723 // Example 2.
1724 x := 2;
1725 y := 2;
```

```
1726 Echo(x, "=", y, ":", x = y);
1727 Echo(x, "!= ", y, ":", x != y);
1728 Echo(x, "< ", y, ":", x < y);
1729 Echo(x, "<= ", y, ":", x <= y);
1730 Echo(x, "> ", y, ":", x > y);
```

```
1731     Echo(x, ">= ", y, ":", x >= y);
```

```
1732 %/mathpiper
```

```
1733     %output,preserve="false"
```

```
1734     Result: True
```

```
1735
```

```
1736     Side Effects:
```

```
1737     2 = 2 :True
```

```
1738     2 != 2 :False
```

```
1739     2 < 2 :False
```

```
1740     2 <= 2 :True
```

```
1741     2 > 2 :False
```

```
1742     2 >= 2 :True
```

```
1743 .    %/output
```

```
1744 %mathpiper
```

```
1745 // Example 3.
```

```
1746 x := 3;
```

```
1747 y := 2;
```

```
1748 Echo(x, "=", y, ":", x = y);
```

```
1749 Echo(x, "!= ", y, ":", x != y);
```

```
1750 Echo(x, "< ", y, ":", x < y);
```

```
1751 Echo(x, "<= ", y, ":", x <= y);
```

```
1752 Echo(x, "> ", y, ":", x > y);
```

```
1753 Echo(x, ">= ", y, ":", x >= y);
```

```
1754 %/mathpiper
```

```
1755     %output,preserve="false"
```

```
1756     Result: True
```

```
1757
```

```
1758     Side Effects:
```

```
1759     3 = 2 :False
```

```
1760     3 != 2 :True
```

```
1761     3 < 2 :False
```

```
1762     3 <= 2 :False
```

```
1763     3 > 2 :True
```

```
1764     3 >= 2 :True
```

```
1765 .    %/output
```

1766 Conditional operators are placed at a lower level of precedence than the other
1767 operators we have covered to this point:

1768 () Parentheses are evaluated from the inside out.

1769 ^ Then exponents are evaluated right to left.

1770 *,%/, Then multiplication, remainder, and division operations are evaluated
1771 left to right.

1772 +, - Then addition and subtraction are evaluated left to right.

1773 =,!=,<,<=,>,>= Finally, conditional operators are evaluated.

1774 **12.2 Predicate Expressions**

1775 Expressions which return either **True** or **False** are called "**predicate**"
1776 expressions. By themselves, predicate expressions are not very useful and they
1777 only become so when they are used with special decision making functions, like
1778 the If() function (which is discussed in the next section).

1779 **12.3 Exercises**

1780 For the following exercises, create a new MathRider worksheet file called
1781 **book_1_section_12a_exercises_<your first name>_<your last name>.mrw.**
1782 (**Note: there are no spaces in this file name**). For example, John Smith's
1783 worksheet would be called:

1784 **book_1_section_12a_exercises_john_smith.mrw.**

1785 After this worksheet has been created, place your answer for each exercise that
1786 requires a fold into its own fold in this worksheet. Place a title attribute in the
1787 start tag of each fold which indicates the exercise the fold contains the solution
1788 to. The folds you create should look similar to this one:

1789 `%mathpiper,title="Exercise 1"`

1790 `//Sample fold.`

1791 `%/mathpiper`

1792 If an exercise uses the MathPiper console instead of a fold, copy the work you
1793 did in the console into the worksheet so it can be saved.

1794 **12.3.1 Exercise 1**

1795 Carefully read all of section 12 up to this point. Evaluate each one of
1796 the examples in the sections you read in the MathPiper worksheet you
1797 created or in the MathPiper console and verify that the results match the
1798 ones in the book. Copy all of the console examples you evaluated into your
1799 worksheet so they will be saved.

12.3.2 Exercise 2

Open a MathPiper session and evaluate the following predicate expressions:

In> 3 = 3

In> 3 = 4

In> 3 < 4

In> 3 != 4

In> -3 < 4

In> 4 >= 4

In> 1/2 < 1/4

In> 15/23 < 122/189

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

In> 1/2 = .5

In> N(1/2) = .5

12.3.3 Exercise 3

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

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

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

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

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

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

1826 that accepts a predicate expression as a parameter can also accept the boolean
1827 values True and False).

1828 The following program uses an **If()** function to determine if the value in variable
1829 number is greater than 5. If number is greater than 5, the program will echo
1830 "Greater" and then "End of program":

```
1831 %mathpiper
1832 number := 6;
1833 If(number > 5, Echo(number, "is greater than 5."));
1834 Echo("End of program.");
1835 %/mathpiper
1836 %output,preserve="false"
1837 Result: True
1838
1839 Side Effects:
1840 6 is greater than 5.
1841 End of program.
1842 . %/output
```

1843 In this program, number has been set to 6 and therefore the expression number
1844 > 5 is **True**. When the **If()** function evaluates the **predicate expression** and
1845 determines it is **True**, it then executes the **first Echo()** function. The **second**
1846 **Echo()** function at the bottom of the program prints "End of program"
1847 regardless of what the If() function does. (**Note: semicolons cannot be placed**
1848 **after expressions which are in function calls.**)

1849 Here is the same program except that **number** has been set to **4** instead of **6**:

```
1850 %mathpiper
1851 number := 4;
1852 If(number > 5, Echo(number, "is greater than 5."));
1853 Echo("End of program.");
1854 %/mathpiper
1855 %output,preserve="false"
1856 Result: True
1857
1858 Side Effects:
1859 End of program.
1860 . %/output
```

1861 This time the expression **number > 4** returns a value of **False** which causes the
1862 **If()** function to not execute the "**then**" expression that was passed to it.

1863 12.4.1 If() Functions Which Include An "Else" Parameter

1864 The second form of the If() function takes a third "**else**" expression which is
1865 executed only if the predicate expression is **False**. This program is similar to the
1866 previous one except an "**else**" expression has been added to it:

```
1867 %mathpiper
1868 x := 4;
1869 If(x > 5, Echo(x, "is greater than 5."), Echo(x, "is NOT greater than 5.));
1870 Echo("End of program.");
1871 %/mathpiper
1872     %output, preserve="false"
1873     Result: True
1874
1875     Side Effects:
1876     4 is NOT greater than 5.
1877     End of program.
1878 .    %/output
```

1879 12.5 Exercises

1880 For the following exercises, create a new MathRider worksheet file called
1881 **book_1_section_12b_exercises_<your first name>_<your last name>.mrw**.
1882 (**Note: there are no spaces in this file name**). For example, John Smith's
1883 worksheet would be called:

1884 **book_1_section_12b_exercises_john_smith.mrw**.

1885 After this worksheet has been created, place your answer for each exercise that
1886 requires a fold into its own fold in this worksheet. Place a title attribute in the
1887 start tag of each fold which indicates the exercise the fold contains the solution
1888 to. The folds you create should look similar to this one:

```
1889 %mathpiper, title="Exercise 1"
1890 //Sample fold.
1891 %/mathpiper
```

1892 If an exercise uses the MathPiper console instead of a fold, copy the work you

1893 did in the console into the worksheet so it can be saved.

1894 **12.5.1 Exercise 1**

1895 Carefully read all of section 12 starting at the end of the previous
1896 exercises and up to this point. Evaluate each one of the examples in the
1897 sections you read in the MathPiper worksheet you created or in the
1898 MathPiper console and verify that the results match the ones in the book.
1899 Copy all of the console examples you evaluated into your worksheet so they
1900 will be saved.

1901 **12.5.2 Exercise 2**

1902 Write a program which uses the RandomInteger() function to simulate the
1903 flipping of a coin (Hint: you can use 1 to represent a head and 0 to
1904 represent a tail.). Use predicate expressions, the If() function, and the
1905 Echo() function to print the string "**The coin came up heads.**" or the string
1906 "**The coin came up tails.**", depending on what the simulated coin flip came
1907 up as when the code was executed.

1908 **12.6 The And(), Or(), & Not() Boolean Functions & Infix Notation**

1909 **12.6.1 And()**

1910 Sometimes a programmer needs to check if two or more expressions are all **True**
1911 and one way to do this is with the **And()** function. The And() function has **two**
1912 **calling formats** (or **notations**) and this is the first one:

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

1913 This calling format is able to accept one or more predicate expressions as input.
1914 If **all** of these expressions returns a value of **True**, the And() function will also
1915 return a **True**. However, if **any** of the expressions return a **False**, then the And()
1916 function will return a **False**. This can be seen in the following example:

```
1917 In> And(True, True)
1918 Result> True
```

```
1919 In> And(True, False)
1920 Result> False
```

```
1921 In> And(False, True)
1922 Result> False
```

```
1923 In> And(True, True, True, True)
1924 Result> True
```

```
1925 In> And(True, True, False, True)
1926 Result> False
```

1927 The second format (or notation) that can be used to call the And() function is
1928 called **infix** notation:

```
expression1 And expression2
```

1929 With **infix** notation, an expression is placed on both sides of the And() function
1930 name instead of being placed inside of parentheses that are next to it:

```
1931 In> True And True
1932 Result> True
```

```
1933 In> True And False
1934 Result> False
```

```
1935 In> False And True
1936 Result> False
```

1937 Infix notation can only accept **two** expressions at a time, but it is often more
1938 convenient to use than function calling notation. The following program also
1939 demonstrates the infix version of the And() function being used:

```
1940 %mathpiper
```

```
1941 a := 7;
1942 b := 9;
```

```
1943 Echo("1: ", a < 5 And b < 10);
1944 Echo("2: ", a > 5 And b > 10);
1945 Echo("3: ", a < 5 And b > 10);
1946 Echo("4: ", a > 5 And b < 10);
```

```
1947 If(a > 5 And b < 10, Echo("These expressions are both true."));
```

```
1948 %/mathpiper
```

```
1949 %output,preserve="false"
1950 Result: True
1951
1952 Side Effects:
1953 1: False
1954 2: False
1955 3: False
1956 4: True
1957 These expressions are both true.
1958 . %/output
```

1959 **12.6.2 Or()**

1960 The Or() function is similar to the And() function in that it has both a function
1961 calling format and an infix calling format and it only works with predicate
1962 expressions. However, instead of requiring that all expressions be **True** in order
1963 to return a **True**, Or() will return a **True** if **one or more expressions are True**.

1964 Here is the function calling format for Or():

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

1965 and this example shows Or() being used with function calling format:

1966 In> Or(True, False)

1967 Result> True

1968 In> Or(False, True)

1969 Result> True

1970 In> Or(False, False)

1971 Result> False

1972 In> Or(False, False, False, False)

1973 Result> False

1974 In> Or(False, True, False, False)

1975 Result> True

1976 The infix notation format for Or() is as follows:

```
expression1 Or expression2
```

1977 and this example shows infix notation being used:

1978 In> True Or False

1979 Result> True

1980 In> False Or True

1981 Result> True

1982 In> False Or False

1983 Result> False

1984 The following program also demonstrates the infix version of the Or() function
1985 being used:

```
1986 %mathpiper
1987 a := 7;
1988 b := 9;
1989 Echo("1: ", a < 5 Or b < 10);
1990 Echo("2: ", a > 5 Or b > 10);
1991 Echo("3: ", a > 5 Or b < 10);
1992 Echo("4: ", a < 5 Or b > 10);
1993 If(a < 5 Or b < 10, Echo("At least one of these expressions is true."));
1994 %/mathpiper
1995 %output,preserve="false"
1996 Result: True
1997
1998 Side Effects:
1999 1: True
2000 2: True
2001 3: True
2002 4: False
2003 At least one of these expressions is true.
2004 . %/output
```

2005 12.6.3 Not() & Prefix Notation

2006 The **Not()** function works with predicate expressions like the And() and Or()
2007 functions do, except it can only accept **one** expression as input. The way Not()
2008 works is that it changes a **True** value to a **False** value and a **False** value to a
2009 **True** value. Here is the Not()'s function calling format:

```
Not(expression)
```

2010 and this example shows Not() being used with function calling format:

```
2011 In> Not(True)
2012 Result> False
2013 In> Not(False)
2014 Result> True
```

2015 Instead of providing an alternative infix calling format like And() and Or() do,
2016 Not()'s second calling format uses **prefix** notation:

```
Not expression
```


2017 Prefix notation looks similar to function notation except no parentheses are used:

```
2018 In> Not True
2019 Result> False
```

```
2020 In> Not False
2021 Result> True
```

2022 Finally, here is a program that also uses the prefix version of Not():

```
2023 %mathpiper
2024 Echo("3 = 3 is ", 3 = 3);
2025 Echo("Not 3 = 3 is ", Not 3 = 3);
2026 %/mathpiper
2027     %output,preserve="false"
2028     Result: True
2029
2030     Side Effects:
2031     3 = 3 is True
2032     Not 3 = 3 is False
2033 .    %/output
```

2034 12.7 Exercises

2035 For the following exercises, create a new MathRider worksheet file called
2036 **book_1_section_12c_exercises_<your first name>_<your last name>.mrw**.
2037 (**Note: there are no spaces in this file name**). For example, John Smith's
2038 worksheet would be called:

2039 **book_1_section_12c_exercises_john_smith.mrw**.

2040 After this worksheet has been created, place your answer for each exercise that
2041 requires a fold into its own fold in this worksheet. Place a title attribute in the
2042 start tag of each fold which indicates the exercise the fold contains the solution
2043 to. The folds you create should look similar to this one:

```
2044 %mathpiper,title="Exercise 1"
2045 //Sample fold.
2046 %/mathpiper
```

2047 If an exercise uses the MathPiper console instead of a fold, copy the work you

2048 did in the console into the worksheet so it can be saved.

2049 **12.7.1 Exercise 1**

2050 Carefully read all of section 12 starting at the end of the previous
2051 exercises and up to this point. Evaluate each one of the examples in the
2052 sections you read in the MathPiper worksheet you created or in the
2053 MathPiper console and verify that the results match the ones in the book.
2054 Copy all of the console examples you evaluated into your worksheet so they
2055 will be saved.

2056 **12.7.2 Exercise 2**

2057 The following program simulates the rolling of two dice and prints a
2058 message if **both** of the two dice come up less than or equal to 3. Create a
2059 similar program which simulates the flipping of two coins and print the
2060 message "Both coins came up heads." if both coins come up heads.

```
2061 %mathpiper
2062 /*
2063    This program simulates the rolling of two dice and prints a message if
2064    both of the two dice come up less than or equal to 3.
2065 */
```

```
2066 dice1 := RandomInteger(6);
2067 dice2 := RandomInteger(6);
```

```
2068 Echo("Dice1: ", dice1, "   Dice2: ", dice2);
2069 NewLine();
```

```
2070 If( dice1 <= 3 And dice2 <= 3, Echo("Both dice came up <= to 3.") );
```

```
2071 %/mathpiper
```

2072 **12.7.3 Exercise 3**

2073 The following program simulates the rolling of two dice and prints a
2074 message if **either** of the two dice come up less than or equal to 3. Create
2075 a similar program which simulates the flipping of two coins and print the
2076 message "At least one coin came up heads." if at least one coin comes up
2077 heads.

```
2078 %mathpiper
2079 /*
2080    This program simulates the rolling of two dice and prints a message if
2081    either of the two dice come up less than or equal to 3.
2082 */
```

```
2083 dice1 := RandomInteger(6);
2084 dice2 := RandomInteger(6);
```

```
2085 Echo("Dice1: ", dice1, "   Dice2: ", dice2);
2086 NewLine();

2087 If( dice1 <= 3 Or dice2 <= 3, Echo("At least one die came up <= 3.") );

2088 %/mathpiper
```

2089 13 The While() Looping Function & Bodied Notation

2090 Many kinds of machines, including computers, derive much of their power from
2091 the principle of **repeated cycling**. **Repeated cycling** in a MathPiper program
2092 means to execute one or more expressions over and over again and this process
2093 is called "**looping**". MathPiper provides a number of ways to implement **loops**
2094 in a program and these ways range from straight-forward to subtle.

2095 We will begin discussing looping in MathPiper by starting with the straight-
2096 forward **While** function. The calling format for the **While** function is as follows:

```
2097 While(predicate)
2098 [
2099     body_expressions
2100 ];
```

2101 The **While** function is similar to the **If** function except it will repeatedly execute
2102 the expressions it contains as long as its "predicate" expression is **True**. As soon
2103 as the predicate expression returns a **False**, the While() function skips the
2104 expressions it contains and execution continues with the expression that
2105 immediately follows the While() function (if there is one).

2106 The expressions which are contained in a While() function are called its "**body**"
2107 and all functions which have body expressions are called "**bodied**" functions. If
2108 a body contains more than one expression then these expressions need to be
2109 placed within a **code block** (code blocks were discussed in an earlier section).
2110 What a function's body is will become clearer after studying some example
2111 programs.

2112 13.1 Printing The Integers From 1 to 10

2113 The following program uses a While() function to print the integers from 1 to 10:

```
2114 %mathpiper
2115 // This program prints the integers from 1 to 10.
2116 /*
2117     Initialize the variable count to 1
2118     outside of the While "loop".
2119 */
2120 count := 1;
2121 While(count <= 10)
2122 [
2123     Echo(count);
```

```
2124
2125     count := count + 1; //Increment count by 1.
2126 1;

2127 %/mathpiper

2128     %output,preserve="false"
2129     Result: True
2130
2131     Side Effects:
2132     1
2133     2
2134     3
2135     4
2136     5
2137     6
2138     7
2139     8
2140     9
2141     10
2142 . %/output
```

2143 In this program, a single variable called **count** is created. It is used to tell the
2144 Echo() function which integer to print and it is also used in the predicate
2145 expression that determines if the While() function should continue to **loop** or not.

2146 When the program is executed, 1 is placed into **count** and then the While()
2147 function is called. The predicate expression **count** <= 10 becomes **1** <= 10
2148 and, since 1 is indeed less than or equal to 10, a value of **True** is returned by the
2149 predicate expression.

2150 The While() function sees that the predicate expression returned a **True** and
2151 therefore it executes all of the expressions inside of its **body** from top to bottom.

2152 The Echo() function prints the current contents of count (which is 1) and then the
2153 expression count := count + 1 is executed.

2154 The expression **count := count + 1** is a standard expression form that is used in
2155 many programming languages. Each time an expression in this form is
2156 evaluated, it **increases the variable it contains by 1**. Another way to describe
2157 the effect this expression has on **count** is to say that it **increments count by 1**.

2158 In this case **count** contains **1** and, after the expression is evaluated, **count**
2159 contains **2**.

2160 After the last expression inside the body of the While() function is executed, the
2161 While() function reevaluates its predicate expression to determine whether it
2162 should continue looping or not. Since **count** is **2** at this point, the predicate
2163 expression returns **True** and the code inside the body of the While() function is
2164 executed again. This loop will be repeated until **count** is incremented to **11** and
2165 the predicate expression returns **False**.

2166 **13.2 Printing The Integers From 1 to 100**

2167 The previous program can be adjusted in a number of ways to achieve different
2168 results. For example, the following program prints the integers from 1 to 100 by
2169 changing the **10** in the predicate expression to **100**. A Write() function is used in
2170 this program so that its output is displayed on the same line until it encounters
2171 the **wrap margin** in MathRider (which can be set in Utilities -> Buffer
2172 Options...).

```
2173 %mathpiper
2174 // Print the integers from 1 to 100.
2175 count := 1;
2176 While(count <= 100)
2177 [
2178     Write(count,,);
2179     count := count + 1; //Increment count by 1.
2180 ];
2181
2182 %/mathpiper
2183     %output,preserve="false"
2184     Result: True
2185
2186     Side Effects:
2187     1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,
2188     24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,
2189     44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,
2190     64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,
2191     84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100
2192 . %/output
```

2193 **13.3 Printing The Odd Integers From 1 To 99**

2194 The following program prints the odd integers from 1 to 99 by changing the
2195 **increment value** in the increment expression from **1** to **2**:

```
2196 %mathpiper
2197 //Print the odd integers from 1 to 99.
2198 x := 1;
2199 While(x <= 100)
2200 [
2201     Write(x,,);
```

```
2202     x := x + 2;    //Increment x by 2.
2203 ];
2204 %/mathpiper
2205     %output,preserve="false"
2206     Result: True
2207
2208     Side Effects:
2209     1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39,41,43,
2210     45,47,49,51,53,55,57,59,61,63,65,67,69,71,73,75,77,79,81,83,
2211     85,87,89,91,93,95,97,99
2212 .    %/output
```

2213 **13.4 Printing The Integers From 1 To 100 In Reverse Order**

2214 Finally, the following program prints the integers from 1 to 100 in reverse order:

```
2215 %mathpiper
2216 //Print the integers from 1 to 100 in reverse order.
2217 x := 100;
2218 While(x >= 1)
2219 [
2220     Write(x,,);
2221     x := x - 1;    //Decrement x by 1.
2222 ];
2223 %/mathpiper
2224     %output,preserve="false"
2225     Result: True
2226
2227     Side Effects:
2228     100,99,98,97,96,95,94,93,92,91,90,89,88,87,86,85,84,83,82,
2229     81,80,79,78,77,76,75,74,73,72,71,70,69,68,67,66,65,64,63,
2230     62,61,60,59,58,57,56,55,54,53,52,51,50,49,48,47,46,45,44,
2231     43,42,41,40,39,38,37,36,35,34,33,32,31,30,29,28,27,26,25,
2232     24,23,22,21,20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,
2233     3,2,1
2234 .    %/output
```

2235 In order to achieve the reverse ordering, this program had to initialize **x** to **100**,
2236 check to see if **x** was **greater than or equal to 1** ($x \geq 1$), and **decrement** **x** by
2237 **subtracting 1 from it** instead of adding 1 to it.

2238 **13.5 Expressions Inside Of Code Blocks Are Indented**

2239 In the programs in the previous sections which use while loops, notice that the
2240 expressions which are inside of the While() function's code block are **indented**.
2241 These expressions do not need to be indented to execute properly, but doing so
2242 makes the program easier to read.

2243 **13.6 Long-Running Loops, Infinite Loops, & Interrupting Execution**

2244 It is easy to create a loop that will execute a **large number of times**, or even **an**
2245 **infinite number of times**, either on purpose or by mistake. When you execute
2246 a program that contains an **infinite loop**, it will run until you tell MathPiper to
2247 **interrupt** its execution. This is done by opening the MathPiper console and then
2248 pressing the "**Stop**" button which it contains. The Stop button is circular and it
2249 has an X on it. (**Note: currently this button only works if MathPiper is**
2250 **executed inside of a %mathpiper fold.**)

2251 Lets experiment with the **Stop** button by executing a program that contains an
2252 infinite loop and then stopping it:

```
2253 %mathpiper
2254 //Infinite loop example program.
2255 x := 1;
2256 While(x < 10)
2257 [
2258     x := 3; //Oops, x is not being incremented!.
2259 ];
2260 %/mathpiper
2261     %output,preserve="false"
2262     Processing...
2263 . %/output
```

2264 Since the contents of x is never changed inside the loop, the expression **x < 10**
2265 always evaluates to **True** which causes the loop to continue looping. Notice that
2266 the %output fold contains the word "**Processing...**" to indicate that the program
2267 is still running the code.

2268 Execute this program now and then interrupt it using the "**Stop**" button. When
2269 the program is interrupted, the %output fold will display the message "**User**
2270 **interrupted calculation**" to indicate that the program was interrupted. After a
2271 program has been interrupted, the program can be edited and then rerun.

2272 **13.7 A Program That Simulates Rolling Two Dice 50 Times**

2273 The following program is larger than the previous programs that have been
2274 discussed in this book, but it is also more interesting and more useful. It uses a
2275 While() loop to simulate the rolling of two dice 50 times and it records how many
2276 times each possible sum has been rolled so that this data can be printed. The
2277 comments in the code explain what each part of the program does. (Remember, if
2278 you copy this program to a MathRider worksheet, you can use **rectangular**
2279 **selection mode** to easily remove the line numbers).

```
2280 %mathpiper
2281 /*
2282     This program simulates rolling two dice 50 times.
2283 */
2284
2285 /*
2286     These variables are used to record how many times
2287     a possible sum of two dice has been rolled. They are
2288     all initialized to 0 before the simulation begins.
2289 */
2289 numberOfTwosRolled := 0;
2290 numberOfThreesRolled := 0;
2291 numberOfFoursRolled := 0;
2292 numberOfFivesRolled := 0;
2293 numberOfSixesRolled := 0;
2294 numberOfSevensRolled := 0;
2295 numberOfEightsRolled := 0;
2296 numberOfNinesRolled := 0;
2297 numberOfTensRolled := 0;
2298 numberOfElevensRolled := 0;
2299 numberOfTwelvesRolled := 0;
2300
2300 //This variable keeps track of the number of the current roll.
2301 roll := 1;
2302
2302 Echo("These are the rolls:");
2303
2303 /*
2304     The simulation is performed inside of this while loop. The number of
2305     times the dice will be rolled can be changed by changing the number 50
2306     which is in the While function's predicate expression.
2307 */
2308 While(roll <= 50)
2309 [
2310     //Roll the dice.
2311     die1 := RandomInteger(6);
2312     die2 := RandomInteger(6);
```

```
2313
2314
2315 //Calculate the sum of the two dice.
2316 rollSum := die1 + die2;
2317
2318
2319 /*
2320 Print the sum that was rolled. Note: if a large number of rolls
2321 is going to be performed (say > 1000), it would be best to comment
2322 out this Write() function so that it does not put too much text
2323 into the output fold.
2324 */
2325 Write(rollSum,,);
2326
2327
2328 /*
2329 These If() functions determine which sum was rolled and then add
2330 1 to the variable which is keeping track of the number of times
2331 that sum was rolled.
2332 */
2333 If(rollSum = 2, numberOfTwosRolled := numberOfTwosRolled + 1);
2334 If(rollSum = 3, numberOfThreesRolled := numberOfThreesRolled + 1);
2335 If(rollSum = 4, numberOfFoursRolled := numberOfFoursRolled + 1);
2336 If(rollSum = 5, numberOfFivesRolled := numberOfFivesRolled + 1);
2337 If(rollSum = 6, numberOfSixesRolled := numberOfSixesRolled + 1);
2338 If(rollSum = 7, numberOfSevensRolled := numberOfSevensRolled + 1);
2339 If(rollSum = 8, numberOfEightsRolled := numberOfEightsRolled + 1);
2340 If(rollSum = 9, numberOfNinesRolled := numberOfNinesRolled + 1);
2341 If(rollSum = 10, numberOfTensRolled := numberOfTensRolled + 1);
2342 If(rollSum = 11, numberOfElevensRolled := numberOfElevensRolled+1);
2343 If(rollSum = 12, numberOfTwelvesRolled := numberOfTwelvesRolled+1);
2344
2345
2346 //Increment the roll variable to the next roll number.
2347 roll := roll + 1;
2348 ];

```



```
2349 //Print the contents of the sum count variables for visual analysis.
2350 NewLine();
2351 NewLine();
2352 Echo("Number of Twos rolled: ", numberOfTwosRolled);
2353 Echo("Number of Threes rolled: ", numberOfThreesRolled);
2354 Echo("Number of Fours rolled: ", numberOfFoursRolled);
2355 Echo("Number of Fives rolled: ", numberOfFivesRolled);
2356 Echo("Number of Sixes rolled: ", numberOfSixesRolled);
2357 Echo("Number of Sevens rolled: ", numberOfSevensRolled);
2358 Echo("Number of Eights rolled: ", numberOfEightsRolled);
2359 Echo("Number of Nines rolled: ", numberOfNinesRolled);
2360 Echo("Number of Tens rolled: ", numberOfTensRolled);
2361 Echo("Number of Elevens rolled: ", numberOfElevensRolled);
2362 Echo("Number of Twelves rolled: ", numberOfTwelvesRolled);
```

```
2363 %/mathpiper
2364 %output,preserve="false"
2365     Result: True
2366
2367     Side effects:
2368     These are the rolls:
2369     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,
2370     12,7,8,12,6,8,10,10,5,9,8,4,5,3,5,7,7,4,6,7,7,5,8,
2371
2372     Number of Twos rolled: 0
2373     Number of Threes rolled: 3
2374     Number of Fours rolled: 6
2375     Number of Fives rolled: 4
2376     Number of Sixes rolled: 6
2377     Number of Sevens rolled: 13
2378     Number of Eights rolled: 6
2379     Number of Nines rolled: 3
2380     Number of Tens rolled: 2
2381     Number of Elevens rolled: 4
2382     Number of Twelves rolled: 3
2383 . %/output
```

2384 13.8 Exercises

2385 For the following exercises, create a new MathRider worksheet file called
2386 **book_1_section_13_exercises_<your first name>_<your last name>.mrw.**
2387 **(Note: there are no spaces in this file name).** For example, John Smith's
2388 worksheet would be called:

2389 **book_1_section_13_exercises_john_smith.mrw.**

2390 After this worksheet has been created, place your answer for each exercise that
2391 requires a fold into its own fold in this worksheet. Place a title attribute in the
2392 start tag of each fold which indicates the exercise the fold contains the solution
2393 to. The folds you create should look similar to this one:

```
2394 %mathpiper,title="Exercise 1"
```

```
2395 //Sample fold.
```

```
2396 %/mathpiper
```

2397 If an exercise uses the MathPiper console instead of a fold, copy the work you
2398 did in the console into the worksheet so it can be saved.

2399 13.8.1 Exercise 1

2400 Carefully read all of section 13 up to this point. Evaluate each one of
2401 the examples in the sections you read in the MathPiper worksheet you
2402 created or in the MathPiper console and verify that the results match the
2403 ones in the book. Copy all of the console examples you evaluated into your
2404 worksheet so they will be saved.

2405 13.8.2 Exercise 2

2406 Create a program which uses a while loop to print the even integers from 2
2407 to 50 inclusive.

2408 13.8.3 Exercise 3

2409 Create a program which prints all the multiples of 5 between 5 and 50
2410 inclusive.

2411 13.8.4 Exercise 4

2412 Create a program which simulates the flipping of a coin 500 times. Print
2413 the number of times the coin came up heads and the number of times it came
2414 up tails after the loop is finished executing.

2415 14 Predicate Functions

2416 A **predicate function** is a function that either returns **True** or **False**. Most
2417 predicate functions in MathPiper have names which begin with "**Is**". For
2418 example, **IsEven()**, **IsOdd()**, **IsInteger()**, etc. The following examples show
2419 some of the predicate functions that are in MathPiper:

```
2420 In> IsEven(4)
```

```
2421 Result> True
```

```
2422 In> IsEven(5)
```

```
2423 Result> False
```

```
2424 In> IsZero(0)
```

```
2425 Result> True
```

```
2426 In> IsZero(1)
```

```
2427 Result> False
```

```
2428 In> IsNegativeInteger(-1)
```

```
2429 Result> True
```

```
2430 In> IsNegativeInteger(1)
```

```
2431 Result> False
```

```
2432 In> IsPrime(7)
```

```
2433 Result> True
```

```
2434 In> IsPrime(100)
```

```
2435 Result> False
```

2436 There is also an **IsBound()** and an **IsUnbound()** function that can be used to
2437 determine whether or not a value is bound to a given variable:

```
2438 In> a
```

```
2439 Result> a
```

```
2440 In> IsBound(a)
```

```
2441 Result> False
```

```
2442 In> a := 1
```

```
2443 Result> 1
```

```
2444 In> IsBound(a)
```

```
2445 Result> True
```

```
2446 In> Clear(a)
```

```
2447 Result> True
```

```
2448 In> a
2449 Result> a
```

```
2450 In> IsBound(a)
2451 Result> False
```

2452 The complete list of predicate functions is contained in the **User**
2453 **Functions/Predicates** node in the MathPiperDocs plugin.

2454 **14.1 Finding Prime Numbers With A Loop**

2455 Predicate functions are very powerful when they are combined with loops
2456 because they can be used to automatically make numerous checks. The
2457 following program uses a while loop to pass the integers 1 through 20 (one at a
2458 time) to the **IsPrime()** function in order to determine which integers are prime
2459 and which integers are not prime:

```
2460 %mathpiper
2461 //Determine which numbers between 1 and 20 (inclusive) are prime.
2462 x := 1;
2463 While(x <= 20)
2464 [
2465     primeStatus := IsPrime(x);
2466     Echo(x, "is prime: ", primeStatus);
2467     x := x + 1;
2468 ];
2469
2470
2471 %/mathpiper
2472 %output,preserve="false"
2473 Result: True
2474
2475 Side Effects:
2476 1 is prime: False
2477 2 is prime: True
2478 3 is prime: True
2479 4 is prime: False
2480 5 is prime: True
2481 6 is prime: False
2482 7 is prime: True
2483 8 is prime: False
2484 9 is prime: False
2485 10 is prime: False
2486 11 is prime: True
2487 12 is prime: False
```

```
2488     13 is prime: True
2489     14 is prime: False
2490     15 is prime: False
2491     16 is prime: False
2492     17 is prime: True
2493     18 is prime: False
2494     19 is prime: True
2495     20 is prime: False
2496 .    %/output
```

2497 This program worked fairly well, but it is limited because it prints a line for each
2498 prime number and also each non-prime number. This means that if large ranges
2499 of integers were processed, enormous amounts of output would be produced.
2500 The following program solves this problem by using an If() function to only print
2501 a number if it is prime:

```
2502 %mathpiper
2503 //Print the prime numbers between 1 and 50 (inclusive).
2504 x := 1;
2505 While(x <= 50)
2506 [
2507     primeStatus := IsPrime(x);
2508     If(primeStatus = True, Write(x,,) );
2509     x := x + 1;
2510 ]
2511
2512 ];
2513 %/mathpiper
2514     %output,preserve="false"
2515     Result: True
2516
2517     Side Effects:
2518     2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,
2519 .    %/output
```

2520 This program is able to process a much larger range of numbers than the
2521 previous one without having its output fill up the text area. However, the
2522 program itself can be shortened by moving the **IsPrime()** function **inside** of the
2523 **If()** function instead of using the **primeStatus** variable to communicate with it:

```
2524 %mathpiper
2525 /*
```

```
2526     Print the prime numbers between 1 and 50 (inclusive).
2527     This is a shorter version which places the IsPrime() function
2528     inside of the If() function instead of using a variable.
2529 */
2530 x := 1;
2531 While(x <= 50)
2532 [
2533     If(IsPrime(x), Write(x,,) );
2534     x := x + 1;
2535 ];
2536 ];
2537 %/mathpiper
2538     %output,preserve="false"
2539     Result: True
2540
2541     Side Effects:
2542     2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,
2543     . %/output
```

2544 14.2 Finding The Length Of A String With The Length() Function

2545 Strings can contain zero or more characters and the **Length()** function can be
2546 used to determine how many characters a string holds:

```
2547 In> s := "Red"
2548 Result> "Red"
2549 In> Length(s)
2550 Result> 3
```

2551 In this example, the string "Red" is assigned to the variable **s** and then **s** is
2552 passed to the **Length()** function. The **Length()** function returned a **3** which
2553 means the string contained **3 characters**.

2554 The following example shows that strings can also be passed to functions
2555 directly:

```
2556 In> Length("Red")
2557 Result> 3
```

2558 An **empty string** is represented by **two double quote marks with no space in**
2559 **between them**. The **length** of an empty string is **0**:


```
2560 In> Length("")
2561 Result> 0
```

2562 **14.3 Converting Numbers To Strings With The String() Function**

2563 Sometimes it is useful to convert a number to a string so that the individual
2564 digits in the number can be analyzed or manipulated. The following example
2565 shows a **number** being converted to a **string** with the **String()** function so that
2566 its **leftmost** and **rightmost** digits can be placed into **variables**:

```
2567 In> number := 523
2568 Result> 523
```

```
2569 In> stringNumber := String(number)
2570 Result> "523"
```

```
2571 In> leftmostDigit := stringNumber[1]
2572 Result> "5"
```

```
2573 In> rightmostDigit := stringNumber[ Length(stringNumber) ]
2574 Result> "3"
```

2575 Notice that the Length() function is used here to determine which character in
2576 **stringNumber** held the **rightmost** digit.

2577 **14.4 Finding Prime Numbers Which End With 7 (And Multi-line Function Calls)**

2579 Now that we have covered how to turn a number into a string, lets use this
2580 ability inside a loop. The following program finds all the **prime numbers**
2581 between **1** and **500** which have a **7 as their rightmost digit**. There are three
2582 important things which are shown in this program:

2583 1) Function calls **can have their parameters placed on more than one**
2584 **line** if the parameters are too long to fit on a **single line**. In this case, a long
2585 code block is being placed inside of an If() function.

2586 2) Code blocks (which are considered to be compound expressions) **cannot**
2587 **have a semicolon placed after them if they are in a function call**. If a
2588 semicolon is placed after this code block, an error will be produced.

2589 3) If() functions can be placed inside of other If() functions in order to make
2590 more complex decisions. This is referred to as **nesting** functions.

2591 When the program is executed, it finds 24 prime numbers which have 7 as their
2592 rightmost digit:

```
2593 %mathpiper
2594 /*
2595     Find all the prime numbers between 1 and 500 which have a 7
2596     as their rightmost digit.
2597 */
2598 x := 1;
2599 While(x <= 500)
2600 [
2601     //Notice how function parameters can be put on more than one line.
2602     If(IsPrime(x),
2603         [
2604             stringVersionOfNumber := String(x);
2605
2606             stringLength := Length(stringVersionOfNumber);
2607
2608             //Notice that If() functions can be placed inside of other
2609             // If() functions.
2610             If(stringVersionOfNumber[stringLength] = "7", Write(x,,) );
2611
2612         ] //Notice that semicolons cannot be placed after code blocks
2613         //which are in function calls.
2614     ); //This is the close parentheses for the outer If() function.
2615     x := x + 1;
2616 ];
2617
2619 %/mathpiper
2620 %output,preserve="false"
2621     Result: True
2622
2623     Side Effects:
2624     7,17,37,47,67,97,107,127,137,157,167,197,227,257,277,307,317,
2625     337,347,367,397,457,467,487,
2626 . %/output
```

2627 It would be nice if we had the ability to store these numbers someplace so that
2628 they could be processed further and this is discussed in the next section.

2629 14.5 Exercises

2630 For the following exercises, create a new MathRider worksheet file called
2631 **book_1_section_14_exercises_<your first name>_<your last name>.mrw.**
2632 **(Note: there are no spaces in this file name).** For example, John Smith's
2633 worksheet would be called:

2634 book_1_section_14_exercises_john_smith.mrw.

2635 After this worksheet has been created, place your answer for each exercise that
2636 requires a fold into its own fold in this worksheet. Place a title attribute in the
2637 start tag of each fold which indicates the exercise the fold contains the solution
2638 to. The folds you create should look similar to this one:

```
2639 %mathpiper,title="Exercise 1"
```

```
2640 //Sample fold.
```

```
2641 %/mathpiper
```

2642 If an exercise uses the MathPiper console instead of a fold, copy the work you
2643 did in the console into the worksheet so it can be saved.

2644 14.5.1 Exercise 1

2645 Carefully read all of section 14 up to this point. Evaluate each one of
2646 the examples in the sections you read in the MathPiper worksheet you
2647 created or in the MathPiper console and verify that the results match the
2648 ones in the book. Copy all of the console examples you evaluated into your
2649 worksheet so they will be saved.

2650 14.5.2 Exercise 2

2651 Write a program which uses a loop to determine how many prime numbers there
2652 are between 1 and 1000. You do not need to print the numbers themselves,
2653 just how many there are.

2654 14.5.3 Exercise 3

2655 Write a program which uses a loop to print all of the prime numbers between
2656 10 and 99 which contain the digit 3 in either their 1's place, or their
2657 10's place, or both places.

2658 15 Lists: Values That Hold Sequences Of Expressions

2659 The **list** value type is designed to hold expressions in an **ordered collection** or
2660 **sequence**. Lists are very flexible and they are one of the most heavily used
2661 value types in MathPiper. Lists can **hold expressions of any type**, they can be
2662 made to **grow and shrink as needed**, and they can be **nested**. Expressions in a
2663 list can be **accessed by their position** in the list (similar to the way that
2664 characters in a string are accessed) and they can also be **replaced by other**
2665 **expressions**.

2666 One way to create a list is by placing zero or more expressions separated by
2667 commas inside of a **pair of braces {}**. In the following example, a list is created
2668 that contains various expressions and then it is assigned to the variable **x**:

```
2669 In> x := {7,42,"Hello",1/2,var}
```

```
2670 Result> {7,42,"Hello",1/2,var}
```

```
2671 In> x
```

```
2672 Result> {7,42,"Hello",1/2,var}
```

2673 The number of expressions in a list can be determined with the **Length()**
2674 function:

```
2675 In> Length({7,42,"Hello",1/2,var})
```

```
2676 Result> 5
```

2677 A single expression in a list can be accessed by placing a set of **brackets []** to
2678 the right of the variable that is bound to the list and then putting the
2679 expression's position number inside of the brackets (**Note: the first expression**
2680 **in the list is at position 1 counting from the left end of the list**):

```
2681 In> x[1]
```

```
2682 Result> 7
```

```
2683 In> x[2]
```

```
2684 Result> 42
```

```
2685 In> x[3]
```

```
2686 Result> "Hello"
```

```
2687 In> x[4]
```

```
2688 Result> 1/2
```

```
2689 In> x[5]
```

```
2690 Result> var
```

2691 The **1st** and **2nd** expressions in this list are **integers**, the **3rd** expression is a

2692 **string**, the **4th** expression is a **rational number** and the **5th** expression is an
2693 **unbound variable**.

2694 Lists can also hold other lists as shown in the following example:

```
2695 In> x := {20, 30, {31, 32, 33}, 40}
```

```
2696 Result> {20,30,{31,32,33},40}
```

```
2697 In> x[1]
```

```
2698 Result> 20
```

```
2699 In> x[2]
```

```
2700 Result> 30
```

```
2701 In> x[3]
```

```
2702 Result> {31,32,33}
```

```
2703 In> x[4]
```

```
2704 Result> 40
```

```
2705
```

2706 The expression in the **3rd** position in the list is another **list** which contains the
2707 integers **31**, **32**, and **33**.

2708 An expression in this second list can be accessed by two **two sets of brackets**:

```
2709 In> x[3][2]
```

```
2710 Result> 32
```

2711 The **3** inside of the first set of brackets accesses the **3rd** member of the **first** list
2712 and the **2** inside of the second set of brackets accesses the **2nd** member of the
2713 **second** list.

2714 **15.1 Append() & Nondestructive List Operations**

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

2715 The **Append()** function adds an expression to the end of a list:

```
2716 In> testList := {21,22,23}
```

```
2717 Result> {21,22,23}
```

```
2718 In> Append(testList, 24)
```

```
2719 Result> {21,22,23,24}
```

2720 However, instead of changing the **original** list, **Append()** creates a **copy** of the
2721 **original** list and appends the expression to the **copy**. This can be confirmed by
2722 evaluating the variable **testList** after the **Append()** function has been called:

```
2723 In> testList
2724 Result> {21,22,23}
```

2725 Notice that the list that is bound to **testList** was not modified by the **Append()**
2726 function. This is called a **nondestructive list operation** and **most MathPiper**
2727 **functions that manipulate lists do so nondestructively**. To have the new list
2728 bound to the variable that is being used, the following technique can be
2729 employed:

```
2730 In> testList := {21,22,23}
2731 Result> {21,22,23}

2732 In> testList := Append(testList, 24)
2733 Result> {21,22,23,24}
```

```
2734 In> testList
2735 Result> {21,22,23,24}
```

2736 After this code has been executed, the new list has indeed been bound to
2737 **testList** as desired.

2738 There are some functions, such as **DestructiveAppend()**, which **do** change the
2739 original list and most of them begin with the word "Destructive". These are
2740 called "destructive functions" and they are advanced functions which are not
2741 covered in this book.

2742 **15.2 Using While Loops With Lists**

2743 Functions that loop can be used to **select each expression in a list in turn** so
2744 that an operation can be performed on these expressions. The following
2745 program uses a while loop to print each of the expressions in a list:

```
2746 %mathpiper
2747 //Print each number in the list.

2748 x := {55,93,40,21,7,24,15,14,82};
2749 y := 1;

2750 While(y <= Length(x))
2751 [
2752     Echo(y, "- ", x[y]);
2753     y := y + 1;
2754 ];

2755 %/mathpiper

2756 %output,preserve="false"
```

```
2757         Result: True
2758
2759         Side Effects:
2760         1 - 55
2761         2 - 93
2762         3 - 40
2763         4 - 21
2764         5 - 7
2765         6 - 24
2766         7 - 15
2767         8 - 14
2768         9 - 82
2769     .    %/output
```

2770 A **loop** can also be used to search through a list. The following program uses a
2771 **While()** function and an **If()** function to search through a list to see if it contains
2772 the number **53**. If 53 is found in the list, a message is printed:

```
2773 %mathpiper

2774 //Determine if 53 is in the list.

2775 testList := {18,26,32,42,53,43,54,6,97,41};
2776 index := 1;

2777 While(index <= Length(testList))
2778 [
2779     If(testList[index] = 53,
2780         Echo("53 was found in the list at position", index));
2781     index := index + 1;
2782 ];

2784 %/mathpiper

2785     %output,preserve="false"
2786     Result: True
2787
2788     Side Effects:
2789     53 was found in the list at position 5
2790 .    %/output
```

2791 When this program was executed, it determined that **53** was present in the list at
2792 position **5**.

2793 15.2.1 Using A While Loop And Append() To Place Values In A List

2794 In an earlier section it was mentioned that it would be nice if we could store a set
2795 of values for later processing and this can be done with a **while loop** and the

2796 **Append()** function. The following program creates an empty list and assigned it
2797 to the variable **primes**. The **while loop** and the **IsPrime()** function is then used
2798 to locate the prime integers between 1 and 50 and the **Append()** function is used
2799 to place them in the list. The last part of the program then prints some
2800 information about the numbers that were placed into the list:

```
2801 %mathpiper
2802 //Place the prime numbers between 1 and 50 (inclusive) into a list.
2803 //Create an empty list.
2804 primes := {};
2805 x := 1;
2806 While(x <= 50)
2807 [
2808     /*
2809         If x is prime, append it to the end of the list and then assign
2810         the new list that is created to the variable 'primes'.
2811     */
2812     If(IsPrime(x), primes := Append(primes, x ) );
2813
2814     x := x + 1;
2815 ];
2816 //Print information about the primes that were found.
2817 Echo("Primes ", primes);
2818 Echo("The number of primes in the list = ", Length(primes) );
2819 Echo("The first number in the list = ", primes[1] );
2820 %/mathpiper
2821     %output,preserve="false"
2822     Result: True
2823
2824     Side Effects:
2825     Primes {2,3,5,7,11,13,17,19,23,29,31,37,41,43,47}
2826     The number of primes in the list = 15
2827     The first number in the list = 2
2828 . %/output
```

2829 The ability to place values into a list with a loop is very powerful and we will be
2830 using this ability throughout the rest of the book.

2831 15.3 Exercises

2832 For the following exercises, create a new MathRider worksheet file called
2833 **book_1_section_15a_exercises_<your first name>_<your last name>.mrw.**

2834 **(Note: there are no spaces in this file name)**. For example, John Smith's
2835 worksheet would be called:

2836 **book_1_section_15a_exercises_john_smith.mrw.**

2837 After this worksheet has been created, place your answer for each exercise that
2838 requires a fold into its own fold in this worksheet. Place a title attribute in the
2839 start tag of each fold which indicates the exercise the fold contains the solution
2840 to. The folds you create should look similar to this one:

2841 `%mathpiper,title="Exercise 1"`

2842 `//Sample fold.`

2843 `%/mathpiper`

2844 If an exercise uses the MathPiper console instead of a fold, copy the work you
2845 did in the console into the worksheet so it can be saved.

2846 **15.3.1 Exercise 1**

2847 Carefully read all of section 15 up to this point. Evaluate each one of
2848 the examples in the sections you read in the MathPiper worksheet you
2849 created or in the MathPiper console and verify that the results match the
2850 ones in the book. Copy all of the console examples you evaluated into your
2851 worksheet so they will be saved.

2852 **15.3.2 Exercise 2**

2853 Create a program that uses a loop and an IsOdd() function to analyze the
2854 following list and then print the number of odd numbers it contains.

2855 `{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}`

2856 **15.3.3 Exercise 3**

2857 Create a program that uses a loop and an IsNegativeNumber() function to
2858 copy all of the negative numbers in the following list into a new list.
2859 Use the variable **negativeNumbers** to hold the new list.

2860 `{36,-29,-33,-6,14,7,-16,-3,-14,37,-38,-8,-45,-21,-26,6,6,38,-20,33,41,-`
2861 `4,24,37,40,29}`

2862 **15.3.4 Exercise 4**

2863 Create a program that uses a loop to analyze the following list and then
2864 print the following information about it:

- 2865 1) The largest number in the list.
2866 2) The smallest number in the list.
2867 3) The sum of all the numbers in the list.

```
2868 {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}
```

2869 **15.4 The ForEach() Looping Function**

2870 The **ForEach()** function uses a **loop** to index through a list like the While()
2871 function does, but it is more flexible and automatic. ForEach() also uses bodied
2872 notation like the While() function and here is its calling format:

```
ForEach(variable, list) body
```

2873 **ForEach()** selects each expression in a list in turn, assigns it to the passed-in
2874 "variable", and then executes the expressions that are inside of "body".
2875 Therefore, body is **executed once for each expression in the list**.

2876 **15.5 Print All The Values In A List Using A ForEach() function**

2877 This example shows how ForEach() can be used to print all of the items in a list:

```
2878 %mathpiper
```

```
2879 //Print all values in a list.
```

```
2880 ForEach(value, {50,51,52,53,54,55,56,57,58,59})
```

```
2881 [  
2882     Echo(value);  
2883 ];
```

```
2884 %/mathpiper
```

```
2885     %output,preserve="false"
```

```
2886     Result: True
```

```
2887  
2888     Side Effects:
```

```
2889     50
```

```
2890     51
```

```
2891     52
```

```
2892     53
```

```
2893     54
```

```
2894     55
```

```
2895     56
```

```
2896     57
```

```
2897     58
```

```
2898     59
```

```
2899 .    %/output
```

2900 **15.6 Calculate The Sum Of The Numbers In A List Using ForEach()**

2901 In previous examples, counting code in the form **x := x + 1** was used to count
2902 how many times a while loop was executed. The following program uses a
2903 **ForEach()** function and a line of code similar to this counter to calculate the
2904 **sum of the numbers in a list:**

```
2905 %mathpiper
2906 /*
2907     This program calculates the sum of the numbers
2908     in a list.
2909 */
2910 //This variable is used to accumulate the sum.
2911 sum := 0;
2912 ForEach(x, {1,2,3,4,5,6,7,8,9,10} )
2913 [
2914     /*
2915         Add the contents of x to the contents of sum
2916         and place the result back into sum.
2917     */
2918     sum := sum + x;
2919
2920     //Print the sum as it is being accumulated.
2921     Write(sum,,);
2922 ];
2923 NewLine(); NewLine();
2924 Echo("The sum of the numbers in the list = ", sum);
2925 %/mathpiper
2926 %output,preserve="false"
2927     Result: True
2928
2929     Side Effects:
2930     1,3,6,10,15,21,28,36,45,55,
2931
2932     The sum of the numbers in the list = 55
2933 . %/output
```

2934 In the above program, the integers **1** through **10** were manually placed into a list
2935 by typing them individually. This method is limited because only a relatively
2936 small number of integers can be placed into a list this way. The following section
2937 discusses an operator which can be used to automatically place a large number
2938 of integers into a list with very little typing.

2939 **15.7 The .. Range Operator**

```
first .. last
```

2940 A programmer often needs to create a list which contains **consecutive integers**
2941 and the **.. "range"** operator can be used to do this. The **first** integer in the list is
2942 placed before the **..** operator and the **last** integer in the list is placed after it
2943 (**Note: there must be a space immediately to the left of the .. operator**
2944 **and a space immediately to the right of it or an error will be generated.**).
2945 Here are some examples:

```
2946 In> 1 .. 10  
2947 Result> {1,2,3,4,5,6,7,8,9,10}
```

```
2948 In> 10 .. 1  
2949 Result> {10,9,8,7,6,5,4,3,2,1}
```

```
2950 In> 1 .. 100  
2951 Result> {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,  
2952         21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,  
2953         38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,  
2954         55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,  
2955         72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,  
2956         89,90,91,92,93,94,95,96,97,98,99,100}
```

```
2957 In> -10 .. 10  
2958 Result> {-10,-9,-8,-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,8,9,10}
```

2959 As these examples show, the **..** operator can generate lists of integers in
2960 ascending order and descending order. It can also generate lists that are very
2961 large and ones that contain negative integers.

2962 Remember, though, if one or both of the spaces around the **..** are omitted, an
2963 error is generated:

```
2964 In> 1..3  
2965 Result>  
2966 Error parsing expression, near token .3.
```

2967 **15.8 Using ForEach() With The Range Operator To Print The Prime**
2968 **Numbers Between 1 And 100**

2969 The following program shows how to use a **ForEach()** function instead of a
2970 **While()** function to print the prime numbers between 1 and 100. Notice that
2971 loops that are implemented with **ForEach()** often require less typing than
2972 their **While()** based equivalents:

```
2973 %mathpiper
2974 /*
2975     This program prints the prime integers between 1 and 100 using
2976     a ForEach() function instead of a While() function. Notice that
2977     the ForEach() version requires less typing than the While()
2978     version.
2979 */
2980 ForEach(x, 1 .. 100)
2981 [
2982     If(IsPrime(x), Write(x,,) );
2983 ];
2984 %/mathpiper
2985     %output,preserve="false"
2986     Result: True
2987
2988     Side Effects:
2989     2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,
2990     73,79,83,89,97,
2991 .    %/output
```

2992 15.8.1 Using ForEach() And The Range Operator To Place The Prime 2993 Numbers Between 1 And 50 Into A List

2994 A ForEach() function can also be used to place values in a list, just the the
2995 While() function can:

```
2996 %mathpiper
2997 /*
2998     Place the prime numbers between 1 and 50 into
2999     a list using a ForEach() function.
3000 */
3001 //Create a new list.
3002 primes := {};
3003 ForEach(number, 1 .. 50)
3004 [
3005     /*
3006         If number is prime, append it to the end of the list and
3007         then assign the new list that is created to the variable
3008         'primes'.
3009     */
3010     If(IsPrime(number), primes := Append(primes, number) );
3011 ];
```

```
3012 //Print information about the primes that were found.
3013 Echo("Primes ", primes);
3014 Echo("The number of primes in the list = ", Length(primes) );
3015 Echo("The first number in the list = ", primes[1] );

3016 %/mathpiper

3017     %output,preserve="false"
3018     Result: True
3019
3020     Side Effects:
3021     Primes {2,3,5,7,11,13,17,19,23,29,31,37,41,43,47}
3022     The number of primes in the list = 15
3023     The first number in the list = 2
3024 .    %/output
```

3025 As can be seen from the above examples, the **ForEach()** function and the **range**
3026 **operator** can do a significant amount of work with very little typing. You will
3027 discover in the next section that MathPiper has functions which are even more
3028 powerful than these two.

3029 15.8.2 Exercises

3030 For the following exercises, create a new MathRider worksheet file called
3031 **book_1_section_15b_exercises_<your first name>_<your last name>.mrw**.
3032 (**Note: there are no spaces in this file name**). For example, John Smith's
3033 worksheet would be called:

3034 **book_1_section_15b_exercises_john_smith.mrw**.

3035 After this worksheet has been created, place your answer for each exercise that
3036 requires a fold into its own fold in this worksheet. Place a title attribute in the
3037 start tag of each fold which indicates the exercise the fold contains the solution
3038 to. The folds you create should look similar to this one:

```
3039 %mathpiper,title="Exercise 1"

3040 //Sample fold.

3041 %/mathpiper
```

3042 If an exercise uses the MathPiper console instead of a fold, copy the work you
3043 did in the console into the worksheet so it can be saved.

3044 15.8.3 Exercise 1

3045 Carefully read all of section 15 starting at the end of the previous
3046 exercises and up to this point. Evaluate each one of the examples in the

3047 sections you read in the MathPiper worksheet you created or in the
3048 MathPiper console and verify that the results match the ones in the book.
3049 Copy all of the console examples you evaluated into your worksheet so they
3050 will be saved.

3051 **15.8.4 Exercise 2**

3052 Create a program that uses a **ForEach()** function and an **IsOdd()** function to
3053 analyze the following list and then print the number of odd numbers it
3054 contains.

3055 {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}

3056 **15.8.5 Exercise 3**

3057 Create a program that uses a **ForEach()** function and an **IsNegativeNumber()**
3058 function to copy all of the negative numbers in the following list into a
3059 new list. Use the variable **negativeNumbers** to hold the new list.

3060 {36,-29,-33,-6,14,7,-16,-3,-14,37,-38,-8,-45,-21,-26,6,6,38,-20,33,41,-
3061 4,24,37,40,29}

3062 **15.8.6 Exercise 4**

3063 Create a program that uses a **ForEach()** function to analyze the following
3064 list and then print the following information about it:

- 3065 1) The largest number in the list.
3066 2) The smallest number in the list.
3067 3) The sum of all the numbers in the list.

3068 {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}

3069 **15.8.7 Exercise 5**

3070 Create a program that uses a **while loop** to make a list that contains **1000**
3071 **random integers** between **1** and **100** inclusive. Then, use a **ForEach()**
3072 function to determine how many integers in the list are **prime** and use an
3073 **Echo()** function to print this total.

3074 **16 Functions & Operators Which Loop Internally**

3075 Looping is such a useful capability that MathPiper has many functions which
3076 loop internally. Now that you have some experience with loops, you can use this
3077 experience to help you imagine how these functions use loops to process the
3078 information that is passed to them.

3079 **16.1 Functions & Operators Which Loop Internally To Process Lists**

3080 This section discusses a number of functions that use loops to process lists.

3081 **16.1.1 TableForm()**

```
TableForm(list)
```

3082 The **TableForm()** function prints the contents of a list in the form of a table.
3083 Each member in the list is printed on its own line and this sometimes makes the
3084 contents of the list easier to read:

```
3085 In> testList := {2,4,6,8,10,12,14,16,18,20}  
3086 Result> {2,4,6,8,10,12,14,16,18,20}
```

```
3087 In> TableForm(testList)  
3088 Result> True  
3089 Side Effects>  
3090 2  
3091 4  
3092 6  
3093 8  
3094 10  
3095 12  
3096 14  
3097 16  
3098 18  
3099 20
```

3100 **16.1.2 Contains()**

3101 The **Contains()** function searches a list to determine if it contains a given
3102 expression. If it finds the expression, it returns **True** and if it doesn't find the
3103 expression, it returns **False**. Here is the calling format for Contains():

```
Contains(list, expression)
```


3104 The following code shows Contains() being used to locate a number in a list:

```
3105 In> Contains({50,51,52,53,54,55,56,57,58,59}, 53)
3106 Result> True
```

```
3107 In> Contains({50,51,52,53,54,55,56,57,58,59}, 75)
3108 Result> False
```

3109 The **Not()** function can also be used with predicate functions like Contains() to
3110 change their results to the opposite truth value:

```
3111 In> Not Contains({50,51,52,53,54,55,56,57,58,59}, 75)
3112 Result> True
```

3113 16.1.3 Find()

```
Find(list, expression)
```

3114 The **Find()** function searches a list for the first occurrence of a given expression.
3115 If the expression is found, the **position of its first occurrence** is returned and
3116 if it is not found, **-1** is returned:

```
3117 In> Find({23, 15, 67, 98, 64}, 15)
3118 Result> 2
```

```
3119 In> Find({23, 15, 67, 98, 64}, 8)
3120 Result> -1
```

3121 16.1.4 Count()

```
Count(list, expression)
```

3122 **Count()** determines the number of times a given expression occurs in a list:

```
3123 In> testList := {a,b,b,c,c,c,d,d,d,d,e,e,e,e,e}
3124 Result> {a,b,b,c,c,c,d,d,d,d,e,e,e,e,e}
```

```
3125 In> Count(testList, c)
3126 Result> 3
```

```
3127 In> Count(testList, e)
3128 Result> 5
```

```
3129 In> Count(testList, z)
3130 Result> 0
```

3131 **16.1.5 Select()**

```
Select(predicate function, list)
```

3132 **Select()** returns a list that contains all the expressions in a list which make a
3133 given predicate function return **True**:

```
3134 In> Select("IsPositiveInteger", {46,87,59,-27,11,86,-21,-58,-86,-52})  
3135 Result> {46,87,59,11,86}
```

3136 In this example, notice that the **name** of the predicate function is passed to
3137 **Select()** in **double quotes**. There are other ways to pass a predicate function to
3138 **Select()** but these are covered in a later section.

3139 Here are some further examples which use the **Select()** function:

```
3140 In> Select("IsOdd", {16,14,82,92,33,74,99,67,65,52})  
3141 Result> {33,99,67,65}
```

```
3142 In> Select("IsEven", {16,14,82,92,33,74,99,67,65,52})  
3143 Result> {16,14,82,92,74,52}
```

```
3144 In> Select("IsPrime", 1 .. 75)  
3145 Result> {2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73}
```

3146 Notice how the third example uses the **..** operator to automatically generate a list
3147 of consecutive integers from 1 to 75 for the **Select()** function to analyze.

3148 **16.1.6 The Nth() Function & The [] Operator**

```
Nth(list, index)
```

3149 The **Nth()** function simply returns the expression which is at a given position in
3150 a list. This example shows the **third** expression in a list being obtained:

```
3151 In> testList := {a,b,c,d,e,f,g}  
3152 Result> {a,b,c,d,e,f,g}
```

```
3153 In> Nth(testList, 3)  
3154 Result> c
```

3155 As discussed earlier, the **[]** operator can also be used to obtain a single
3156 expression from a list:

```
3157 In> testList[3]
3158 Result> c
```

3159 The **[]** operator can even obtain a single expression directly from a list without
3160 needing to use a variable:

```
3161 In> {a,b,c,d,e,f,g}[3]
3162 Result> c
```

3163 **16.1.7 The : Prepend Operator**

```
expression : list
```

3164 The prepend operator is a colon **:** and it can be used to add an expression to the
3165 beginning of a list:

```
3166 In> testList := {b,c,d}
3167 Result> {b,c,d}

3168 In> testList := a:testList
3169 Result> {a,b,c,d}
```

3170 **16.1.8 Concat()**

```
Concat(list1, list2, ...)
```

3171 The Concat() function is short for "concatenate" which means to join together
3172 sequentially. It takes two or more lists and joins them together into a single
3173 larger list:

```
3174 In> Concat({a,b,c}, {1,2,3}, {x,y,z})
3175 Result> {a,b,c,1,2,3,x,y,z}
```

3176 **16.1.9 Insert(), Delete(), & Replace()**

```
Insert(list, index, expression)
```

```
Delete(list, index)
```

```
Replace(list, index, expression)
```

3177 **Insert()** inserts an expression into a list at a given index, **Delete()** deletes an
3178 expression from a list at a given index, and **Replace()** replaces an expression in
3179 a list at a given index with another expression:

```
3180 In> testList := {a,b,c,d,e,f,g}
3181 Result> {a,b,c,d,e,f,g}

3182 In> testList := Insert(testList, 4, 123)
3183 Result> {a,b,c,123,d,e,f,g}

3184 In> testList := Delete(testList, 4)
3185 Result> {a,b,c,d,e,f,g}

3186 In> testList := Replace(testList, 4, xxx)
3187 Result> {a,b,c,xxx,e,f,g}
```

3188 16.1.10 Take()

```
Take(list, amount)
Take(list, -amount)
Take(list, {begin_index,end_index})
```

3189 **Take()** obtains a sublist from the **beginning** of a list, the **end** of a list, or the
3190 **middle** of a list. The expressions in the list that are not taken are discarded.

3191 A **positive** integer passed to Take() indicates how many expressions should be
3192 taken from the **beginning** of a list:

```
3193 In> testList := {a,b,c,d,e,f,g}
3194 Result> {a,b,c,d,e,f,g}

3195 In> Take(testList, 3)
3196 Result> {a,b,c}
```

3197 A **negative** integer passed to Take() indicates how many expressions should be
3198 taken from the **end** of a list:

```
3199 In> Take(testList, -3)
3200 Result> {e,f,g}
```

3201 Finally, if a **two member list** is passed to Take() it indicates the **range** of
3202 expressions that should be taken from the **middle** of a list. The **first** value in the
3203 passed-in list specifies the **beginning** index of the range and the **second** value
3204 specifies its **end**:

```
3205 In> Take(testList, {3,5})
3206 Result> {c,d,e}
```

3207 **16.1.11 Drop()**

```
Drop(list, index)
Drop(list, -index)
Drop(list, {begin_index,end_index})
```

3208 **Drop()** does the opposite of Take() in that it **drops** expressions from the
3209 **beginning** of a list, the **end** of a list, or the **middle** of a list and **returns a list**
3210 **which contains the remaining expressions.**

3211 A **positive** integer passed to Drop() indicates how many expressions should be
3212 dropped from the **beginning** of a list:

```
3213 In> testList := {a,b,c,d,e,f,g}
3214 Result> {a,b,c,d,e,f,g}
```

```
3215 In> Drop(testList, 3)
3216 Result> {d,e,f,g}
```

3217 A **negative** integer passed to Drop() indicates how many expressions should be
3218 dropped from the **end** of a list:

```
3219 In> Drop(testList, -3)
3220 Result> {a,b,c,d}
```

3221 Finally, if a **two member list** is passed to Drop() it indicates the **range** of
3222 expressions that should be dropped from the **middle** of a list. The **first** value in
3223 the passed-in list specifies the **beginning** index of the range and the **second**
3224 value specifies its **end**:

```
3225 In> Drop(testList, {3,5})
3226 Result> {a,b,f,g}
```

3227 **16.1.12 FillList()**

```
FillList(expression, length)
```

3228 The FillList() function simply creates a list which is of size "length" and fills it
3229 with "length" copies of the given expression:

```
3230 In> FillList(a, 5)
3231 Result> {a,a,a,a,a}
```

```
3232 In> FillList(42,8)
3233 Result> {42,42,42,42,42,42,42,42}
```

3234 **16.1.13 RemoveDuplicates()**

```
RemoveDuplicates(list)
```

3235 **RemoveDuplicates()** removes any duplicate expressions that are contained in a
3236 list:

```
3237 In> testList := {a,a,b,c,c,b,b,a,b,c,c}  
3238 Result> {a,a,b,c,c,b,b,a,b,c,c}
```

```
3239 In> RemoveDuplicates(testList)  
3240 Result> {a,b,c}
```

3241 **16.1.14 Reverse()**

```
Reverse(list)
```

3242 **Reverse()** reverses the order of the expressions in a list:

```
3243 In> testList := {a,b,c,d,e,f,g,h}  
3244 Result> {a,b,c,d,e,f,g,h}
```

```
3245 In> Reverse(testList)  
3246 Result> {h,g,f,e,d,c,b,a}
```

3247 **16.1.15 Partition()**

```
Partition(list, partition_size)
```

3248 The **Partition()** function breaks a list into sublists of size "partition_size":

```
3249 In> testList := {a,b,c,d,e,f,g,h}  
3250 Result> {a,b,c,d,e,f,g,h}
```

```
3251 In> Partition(testList, 2)  
3252 Result> {{a,b},{c,d},{e,f},{g,h}}
```

3253 If the partition_size does not divide the length of the list **evenly**, the remaining
3254 elements are discarded:

```
3255 In> Partition(testList, 3)  
3256 Result> {{h,b,c},{d,e,f}}
```

3257 The number of elements that Partition() will discard can be calculated by
3258 dividing the length of a list by the partition size and obtaining the **remainder**:

```
3259 In> Length(testList) % 3  
3260 Result> 2
```

3261 Remember that % is the remainder operator. It divides two integers and returns
3262 their remainder.

3263 16.1.16 Table()

```
Table(expression, variable, begin_value, end_value, step_amount)
```

3264 The Table() function creates a list of values by doing the following:

- 3265 1) Generating a sequence of values between a "begin_value" and an
3266 "end_value" with each value being incremented by the "step_amount".
- 3267 2) Placing each value in the sequence into the specified "variable", one value
3268 at a time.
- 3269 3) Evaluating the defined "expression" (which contains the defined "variable")
3270 for each value, one at a time.
- 3271 4) Placing the result of each "expression" evaluation into the result list.

3272 This example generates a list which contains the integers 1 through 10:

```
3273 In> Table(x, x, 1, 10, 1)  
3274 Result> {1,2,3,4,5,6,7,8,9,10}
```

3275 Notice that the expression in this example is simply the variable 'x' itself with no
3276 other operations performed on it.

3277 The following example is similar to the previous one except that its expression
3278 multiplies 'x' by 2:

```
3279 In> Table(x*2, x, 1, 10, 1)  
3280 Result> {2,4,6,8,10,12,14,16,18,20}
```

3281 Lists which contain decimal values can also be created by setting the
3282 "step_amount" to a decimal:

```
3283 In> Table(x, x, 0, 1, .1)  
3284 Result> {0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1}
```

3285 **16.1.17 HeapSort()**

```
HeapSort(list, compare)
```

3286 **HeapSort()** sorts the elements of **list** into the order indicated by **compare** with
3287 **compare** typically being the **less than** operator "<" or the **greater than**
3288 operator ">":

```
3289 In> HeapSort({4,7,23,53,-2,1}, "<");  
3290 Result: {-2,1,4,7,23,53}
```

```
3291 In> HeapSort({4,7,23,53,-2,1}, ">");  
3292 Result: {53,23,7,4,1,-2}
```

```
3293 In> HeapSort({1/2,3/5,7/8,5/16,3/32}, "<")  
3294 Result: {3/32,5/16,1/2,3/5,7/8}
```

```
3295 In> HeapSort({.5,3/5,.76,5/16,3/32}, "<")  
3296 Result: {3/32,5/16,.5,3/5,.76}
```

3297 **16.2 Functions That Work With Integers**

3298 This section discusses various functions which work with integers. Some of
3299 these functions also work with non-integer values and their use with non-
3300 integers is discussed in other sections.

3301 **16.2.1 RandomIntegerVector()**

```
RandomIntegerVector(length, lowest_possible, highest_possible)
```

3302 A vector is a list that does not contain other lists. **RandomIntegerVector()**
3303 creates a list of size "length" that contains random integers that are no lower
3304 than "lowest_possible" and no higher than "highest possible". The following
3305 example creates **10** random integers between **1** and **99** inclusive:

```
3306 In> RandomIntegerVector(10, 1, 99)  
3307 Result> {73,93,80,37,55,93,40,21,7,24}
```

3308 **16.2.2 Max() & Min()**

```
Max(value1, value2)  
Max(list)
```

3309 If two values are passed to **Max()**, it determines which one is larger:

```
3310 In> Max(10, 20)
```


3311 `Result> 20`

3312 If a list of values are passed to `Max()`, it finds the largest value in the list:

3313 `In> testList := RandomIntegerVector(10, 1, 99)`

3314 `Result> {73,93,80,37,55,93,40,21,7,24}`

3315 `In> Max(testList)`

3316 `Result> 93`

3317 The **Min()** function is the opposite of the `Max()` function.

```
Min(value1, value2)
Min(list)
```

3318 If two values are passed to `Min()`, it determines which one is smaller:

3319 `In> Min(10, 20)`

3320 `Result> 10`

3321 If a list of values are passed to `Min()`, it finds the smallest value in the list:

3322 `In> testList := RandomIntegerVector(10, 1, 99)`

3323 `Result> {73,93,80,37,55,93,40,21,7,24}`

3324 `In> Min(testList)`

3325 `Result> 7`

3326 **16.2.3 Div() & Mod()**

```
Div(dividend, divisor)
Mod(dividend, divisor)
```

3327 **Div()** stands for "divide" and determines the whole number of times a divisor
3328 goes into a dividend:

3329 `In> Div(7, 3)`

3330 `Result> 2`

3331 **Mod()** stands for "modulo" and it determines the remainder that results when a
3332 dividend is divided by a divisor:

3333 `In> Mod(7,3)`

3334 `Result> 1`

3335 The remainder/modulo operator **%** can also be used to calculate a remainder:

```
3336 In> 7 % 2
3337 Result> 1
```

3338 **16.2.4 Gcd()**

```
Gcd(value1, value2)
Gcd(list)
```

3339 GCD stands for Greatest Common Divisor and the **Gcd()** function determines the
3340 greatest common divisor of the values that are passed to it.

3341 If two integers are passed to Gcd(), it calculates their greatest common divisor:

```
3342 In> Gcd(21, 56)
3343 Result> 7
```

3344 If a list of integers are passed to Gcd(), it finds the greatest common divisor of all
3345 the integers in the list:

```
3346 In> Gcd({9, 66, 123})
3347 Result> 3
```

3348 **16.2.5 Lcm()**

```
Lcm(value1, value2)
Lcm(list)
```

3349 LCM stands for Least Common Multiple and the **Lcm()** function determines the
3350 least common multiple of the values that are passed to it.

3351 If two integers are passed to Lcm(), it calculates their least common multiple:

```
3352 In> Lcm(14, 8)
3353 Result> 56
```

3354 If a list of integers are passed to Lcm(), it finds the least common multiple of all
3355 the integers in the list:

```
3356 In> Lcm({3, 7, 9, 11})
3357 Result> 693
```

3358 **16.2.6 Sum()**

```
Sum(list)
```

3359 **Sum()** can find the sum of a list that is passed to it:

3360 In> testList := RandomIntegerVector(10,1,99)

3361 Result> {73,93,80,37,55,93,40,21,7,24}

3362 In> Sum(testList)

3363 Result> 523

3364 In> testList := 1 .. 10

3365 Result> {1,2,3,4,5,6,7,8,9,10}

3366 In> Sum(testList)

3367 Result> 55

3368 **16.2.7 Product()**

```
Product(list)
```

3369 This function has two calling formats, only one of which is discussed here.

3370 **Product(list)** multiplies all the expressions in a list together and returns their
3371 product:

3372 In> Product({1,2,3})

3373 Result> 6

3374 **16.3 Exercises**3375 For the following exercises, create a new MathRider worksheet file called
3376 **book_1_section_16_exercises_<your first name>_<your last name>.mrw.**
3377 (**Note: there are no spaces in this file name**). For example, John Smith's
3378 worksheet would be called:3379 **book_1_section_16_exercises_john_smith.mrw.**3380 After this worksheet has been created, place your answer for each exercise that
3381 requires a fold into its own fold in this worksheet. Place a title attribute in the
3382 start tag of each fold which indicates the exercise the fold contains the solution
3383 to. The folds you create should look similar to this one:

3384 %mathpiper,title="Exercise 1"

3385 //Sample fold.

3386 [%/mathpiper](#)

3387 If an exercise uses the MathPiper console instead of a fold, copy the work you
3388 did in the console into the worksheet so it can be saved.

3389 **16.3.1 Exercise 1**

3390 Carefully read all of section 16 up to this point. Evaluate each one of
3391 the examples in the sections you read in the MathPiper worksheet you
3392 created or in the MathPiper console and verify that the results match the
3393 ones in the book. Copy all of the console examples you evaluated into your
3394 worksheet so they will be saved.

3395 **16.3.2 Exercise 2**

3396 Create a program that uses **RandomIntegerVector()** to create a 100 member
3397 list that contains random integers between 1 and 5 inclusive. Use **Count()**
3398 to determine how many of each digit 1-5 are in the list and then print this
3399 information. Hint: you can use the **HeapSort()** function to sort the
3400 generated list to make it easier to check if your program is counting
3401 correctly.

3402 **16.3.3 Exercise 3**

3403 Create a program that uses **RandomIntegerVector()** to create a 100 member
3404 list that contains random integers between 1 and 50 inclusive and use
3405 **Contains()** to determine if the number 25 is in the list. Print "25 was in
3406 the list." if 25 was found in the list and "25 was not in the list." if it
3407 wasn't found.

3408 **16.3.4 Exercise 4**

3409 Create a program that uses **RandomIntegerVector()** to create a 100 member
3410 list that contains random integers between 1 and 50 inclusive and use
3411 **Find()** to determine if the number 10 is in the list. Print the position of
3412 10 if it was found in the list and "10 was not in the list." if it wasn't
3413 found.

3414 **16.3.5 Exercise 5**

3415 Create a program that uses **RandomIntegerVector()** to create a 100 member
3416 list that contains random integers between 0 and 3 inclusive. Use **Select()**
3417 with the **IsNonZeroInteger()** predicate function to obtain all of the nonzero
3418 integers in this list.

3419 **16.3.6 Exercise 6**

3420 Create a program that uses **Table()** to obtain a list which contains the
3421 squares of the integers between 1 and 10 inclusive.

3422 17 Nested Loops

3423 Now that you have seen how to solve problems with single loops, it is time to
3424 discuss what can be done when a loop is placed inside of another loop. A loop
3425 that is placed **inside** of another loop it is called a **nested loop** and this nesting
3426 can be extended to numerous levels if needed. This means that loop 1 can have
3427 loop 2 placed inside of it, loop 2 can have loop 3 placed inside of it, loop 3 can
3428 have loop 4 placed inside of it, and so on.

3429 Nesting loops allows the programmer to accomplish an enormous amount of
3430 work with very little typing.

3431 17.1 Generate All The Combinations That Can Be Entered Into A Two Digit 3432 Wheel Lock Using Two Nested Loops



3433 The following program generates all the combinations that can be entered into a
3434 two digit wheel lock. It uses a nested loop to accomplish this with the "**inside**"
3435 nested loop being used to generate **one's place** digits and the "**outside**" loop
3436 being used to generate **ten's place** digits.

```
3437 %mathpiper
3438 /*
3439  Generate all the combinations can be entered into a two
3440  digit wheel lock.
3441  */
3442 combinations := {};
3443 ForEach(digit1, 0 .. 9) //This loop is called the "outside" loop.
```

```
3444 [
3445     ForEach(digit2, 0 .. 9)//This loop is called the "inside" loop.
3446     [
3447         combinations := Append(combinations, {digit1, digit2});
3448     ];
3449 ];

3450 Echo(TableForm(combinations));

3451 %/mathpiper

3452     %output,preserve="false"
3453     Result: True
3454
3455     Side Effects:
3456     {0,0}
3457     {0,1}
3458     {0,2}
3459     {0,3}
3460     {0,4}
3461     {0,5}
3462     {0,6}
3463     .
3464     . //The middle of the list has not been shown.
3465     .
3466     {9,3}
3467     {9,4}
3468     {9,5}
3469     {9,6}
3470     {9,7}
3471     {9,8}
3472     {9,9}
3473     True
3474 . %/output
```

3475 The relationship between the outside loop and the inside loop is interesting
3476 because each time the **outside loop cycles once**, the **inside loop cycles 10**
3477 **times**. Study this program carefully because nested loops can be used to solve a
3478 wide range of problems and therefore understanding how they work is
3479 important.

3480 17.2 Exercises

3481 For the following exercises, create a new MathRider worksheet file called
3482 **book_1_section_17_exercises_<your first name>_<your last name>.mrw**.
3483 (**Note: there are no spaces in this file name**). For example, John Smith's
3484 worksheet would be called:

3485 **book_1_section_17_exercises_john_smith.mrw**.

3486 After this worksheet has been created, place your answer for each exercise that
3487 requires a fold into its own fold in this worksheet. Place a title attribute in the
3488 start tag of each fold which indicates the exercise the fold contains the solution
3489 to. The folds you create should look similar to this one:

3490 `%mathpiper,title="Exercise 1"`

3491 `//Sample fold.`

3492 `%/mathpiper`

3493 If an exercise uses the MathPiper console instead of a fold, copy the work you
3494 did in the console into the worksheet so it can be saved.

3495 **17.2.1 Exercise 1**

3496 Carefully read all of section 17 up to this point. Evaluate each one of
3497 the examples in the sections you read in the MathPiper worksheet you
3498 created or in the MathPiper console and verify that the results match the
3499 ones in the book. Copy all of the console examples you evaluated into your
3500 worksheet so they will be saved.

3501 **17.2.2 Exercise 2**

3502 Create a program that will generate all of the combinations that can be
3503 entered into a three digit wheel lock. (Hint: a triple nested loop can be
3504 used to accomplish this.)

3505 18 User Defined Functions

3506 In computer programming, a **function** is a named section of code that can be
3507 **called** from other sections of code. **Values** can be sent to a function for
3508 processing as part of the **call** and a function always returns a value as its result.
3509 A function can also generate side effects when it is called and side effects have
3510 been covered in earlier sections.

3511 The values that are sent to a function when it is called are called **arguments** or
3512 **actual parameters** and a function can accept 0 or more of them. These
3513 arguments are placed within parentheses.

3514 MathPiper has many predefined functions (some of which have been discussed in
3515 previous sections) but users can create their own functions too. The following
3516 program creates a function called **addNums()** which takes two numbers as
3517 arguments, adds them together, and returns their sum back to the calling code
3518 as a result:

```
3519 In> addNums(num1,num2) := num1 + num2
3520 Result> True
```

3521 This line of code defined a new function called **addNums** and specified that it
3522 will accept two values when it is called. The **first** value will be placed into the
3523 variable **num1** and the **second** value will be placed into the variable **num2**.

3524 Variables like num1 and num2 which are used in a function to accept values from
3525 calling code are called **formal parameters**. **Formal parameter variables** are
3526 used inside a function to process the **values/actual parameters/arguments**
3527 that were placed into them by the calling code.

3528 The code on the **right side** of the **assignment operator** is **bound** to the
3529 function name "**addNums**" and it is executed each time **addNums()** is called.
3530 The following example shows the new **addNums()** function being called multiple
3531 times with different values being passed to it:

```
3532 In> addNums(2,3)
3533 Result> 5
```

```
3534 In> addNums(4,5)
3535 Result> 9
```

```
3536 In> addNums(9,1)
3537 Result> 10
```

3538 Notice that, unlike the functions that come with MathPiper, we chose to have this
3539 function's name start with a **lower case letter**. We could have had addNums()
3540 begin with an upper case letter but it is a **convention** in MathPiper for **user**

3541 **defined function names to begin with a lower case letter to distinguish**
3542 **them from the functions that come with MathPiper.**

3543 The values that are returned from user defined functions can also be assigned to
3544 variables. The following example uses a %mathpiper fold to define a function
3545 called **evenIntegers()** and then this function is used in the MathPiper console:

```
3546 %mathpiper
3547 evenIntegers(endInteger) :=
3548 [
3549     resultList := {};
3550
3551     x := 2;
3552     While(x <= endInteger)
3553     [
3554         resultList := Append(resultList, x);
3555
3556         x := x + 2;
3557     ];
3558     /*
3559     The result of the last expression which is executed in a function
3560     is the result that the function returns to the caller. In this case,
3561     resultList is purposely being executed last so that its contents are
3562     returned to the caller.
3563     */
3564     resultList;
3565 ];
3566 %/mathpiper
3567 %output,preserve="false"
3568 Result: True
3569 . %/output
3570 In> a := evenIntegers(10)
3571 Result> {2,4,6,8,10}
3572 In> Length(a)
3573 Result> 5
```

3574 The function **evenIntegers()** returns a list which contains all the even integers
3575 from 2 up through the value that was passed into it. The fold was first executed
3576 in order to define the **evenIntegers()** function and make it ready for use. The
3577 **evenIntegers()** function was then called from the MathPiper console and 10
3578 was passed to it.

3579 After the function was finished executing, it returned a list of even integers as a

3580 result and this result was assigned to the variable 'a'. We then passed the list
3581 that was assigned to 'a' to the **Length()** function in order to determine its size.

3582 **18.1 Global Variables, Local Variables, & Local()**

3583 The new **evenIntegers()** function seems to work well, but there is a problem.
3584 The variables 'x' and **resultList** were defined inside the function as **global**
3585 **variables** which means they are accessible from anywhere, including from
3586 within other functions, within other folds (as shown here):

```
3587 %mathpiper
3588 Echo(x, ",", resultList);
3589 %/mathpiper
3590     %output,preserve="false"
3591     Result: True
3592
3593     Side Effects:
3594     12 , {2,4,6,8,10}
3595 .    %/output
```

3596 and from within the MathPiper console:

```
3597 In> x
3598 Result> 12
3599 In> resultList
3600 Result> {2,4,6,8,10}
```

3601 **Using global variables inside of functions is usually not a good idea**
3602 because code in other functions and folds might already be using (or will use) the
3603 same variable names. Global variables which have the same name are the same
3604 variable. When one section of code changes the value of a given global variable,
3605 the value is changed everywhere that variable is used and this will eventually
3606 cause problems.

3607 In order to prevent errors being caused by global variables having the same
3608 name, a function named **Local()** can be called inside of a function to define what
3609 are called **local variables**. A **local variable** is only accessible inside the
3610 function it has been defined in, even if it has the same name as a global variable.
3611 The following example shows a second version of the **evenIntegers()** function
3612 which uses **Local()** to make 'x' and **resultList** local variables:

```
3613 %mathpiper
3614 /*
3615  This version of evenIntegers() uses Local() to make
3616  x and resultList local variables
3617  */
3618 evenIntegers(endInteger) :=
3619 [
3620     Local(x,resultList);
3621     resultList := {};
3622
3623     x := 2;
3624
3625     While(x <= endInteger)
3626     [
3627         resultList := Append(resultList, x);
3628         x := x + 2;
3629     ];
3630
3631     /*
3632     The result of the last expression which is executed in a function
3633     is the result that the function returns to the caller.  In this case,
3634     resultList is purposely being executed last so that its contents are
3635     returned to the caller.
3636     */
3637     resultList;
3638 ];
3639 %/mathpiper
3640 %output,preserve="false"
3641 Result: True
3642 . %/output
```

3643 We can verify that '**x**' and **resultList** are now local variables by first clearing
3644 them, calling **evenIntegers()**, and then seeing what '**x**' and **resultList** contain:

```
3645 In> Clear(x, resultList)
3646 Result> True
3647 In> evenIntegers(10)
3648 Result> {2,4,6,8,10}
3649 In> x
3650 Result> x
3651 In> resultList
3652 Result> resultList
```

3653 18.2 Exercises

3654 For the following exercises, create a new MathRider worksheet file called
3655 **book_1_section_18_exercises_<your first name>_<your last name>.mrw.**
3656 **(Note: there are no spaces in this file name).** For example, John Smith's
3657 worksheet would be called:

3658 **book_1_section_18_exercises_john_smith.mrw.**

3659 After this worksheet has been created, place your answer for each exercise that
3660 requires a fold into its own fold in this worksheet. Place a title attribute in the
3661 start tag of each fold which indicates the exercise the fold contains the solution
3662 to. The folds you create should look similar to this one:

```
3663 %mathpiper,title="Exercise 1"
```

```
3664 //Sample fold.
```

```
3665 %/mathpiper
```

3666 If an exercise uses the MathPiper console instead of a fold, copy the work you
3667 did in the console into the worksheet so it can be saved.

3668 18.2.1 Exercise 1

3669 Carefully read all of section 18 up to this point. Evaluate each one of
3670 the examples in the sections you read in the MathPiper worksheet you
3671 created or in the MathPiper console and verify that the results match the
3672 ones in the book. Copy all of the console examples you evaluated into your
3673 worksheet so they will be saved.

3674 18.2.2 Exercise 2

3675 Create a function called **tenOddIntegers()** which returns a list which
3676 contains 10 random odd integers between 1 and 99 inclusive.

3677 18.2.3 Exercise 3

3678 Create a function called **convertStringToList(string)** which takes a string
3679 as a parameter and returns a list which contains all of the characters in
3680 the string. Here is an example of how the function should work:

```
3681 In> convertStringToList("Hello friend!")  
3682 Result> {"H","e","l","l","o"," ","f","r","i","e","n","d","!"}
```

```
3683 In> convertStringToList("Computer Algebra System")  
3684 Result> {"C","o","m","p","u","t","e","r"," ","A","l","g","e","b","r","a","  
3685 ","S","y","s","t","e","m"}
```

3686 19 Miscellaneous topics

3687 19.1 Incrementing And Decrementing Variables With The ++ And -- 3688 Operators

3689 Up until this point we have been adding 1 to a variable with code in the form of **x**
3690 **:= x + 1** and subtracting 1 from a variable with code in the form of **x := x - 1**.
3691 Another name for **adding** 1 to a variable is **incrementing** it and **decrementing**
3692 a variable means to **subtract** 1 from it. Now that you have had some experience
3693 with these longer forms, it is time to show you shorter versions of them.

3694 19.1.1 Incrementing Variables With The ++ Operator

3695 The number 1 can be added to a variable by simply placing the ++ operator after
3696 it like this:

```
3697 In> x := 1  
3698 Result: 1
```

```
3699 In> x++;  
3700 Result: True
```

```
3701 In> x  
3702 Result: 2
```

3703 Here is a program that uses the ++ operator to increment a loop index variable:

```
3704 %mathpiper  
3705 count := 1;  
3706 While(count <= 10)  
3707 [  
3708     Echo(count);  
3709     count++; //The ++ operator increments the count variable.  
3710 ];  
3712 %/mathpiper  
3713 %output,preserve="false"  
3714 Result: True  
3715  
3716 Side Effects:  
3717 1  
3718 2
```

```
3719      3
3720      4
3721      5
3722      6
3723      7
3724      8
3725      9
3726     10
3727 .    %/output
```

3728 19.1.2 Decrementing Variables With The -- Operator

3729 The number 1 can be subtracted from a variable by simply placing the --
3730 operator after it like this:

```
3731 In> x := 1
3732 Result: 1

3733 In> x--;
3734 Result: True

3735 In> x
3736 Result: 0
```

3737 Here is a program that uses the -- operator to decrement a loop index variable:

```
3738 %mathpiper

3739 count := 10;

3740 While(count >= 1)
3741 [
3742     Echo(count);
3743
3744     count--; //The -- operator decrements the count variable.
3745 ];

3746 %/mathpiper

3747 %output,preserve="false"
3748 Result: True
3749
3750 Side Effects:
3751 10
3752 9
3753 8
3754 7
3755 6
3756 5
```

```
3757         4
3758         3
3759         2
3760         1
3761     .    %/output
```

3762 19.2 Exercises

3763 For the following exercises, create a new MathRider worksheet file called
3764 **book_1_section_19_exercises_<your first name>_<your last name>.mrw.**
3765 (**Note: there are no spaces in this file name**). For example, John Smith's
3766 worksheet would be called:

3767 **book_1_section_19_exercises_john_smith.mrw.**

3768 After this worksheet has been created, place your answer for each exercise that
3769 requires a fold into its own fold in this worksheet. Place a title attribute in the
3770 start tag of each fold which indicates the exercise the fold contains the solution
3771 to. The folds you create should look similar to this one:

```
3772 %mathpiper,title="Exercise 1"
```

```
3773 //Sample fold.
```

```
3774 %/mathpiper
```

3775 If an exercise uses the MathPiper console instead of a fold, copy the work you
3776 did in the console into the worksheet so it can be saved.

3777 19.2.1 Exercise 1

3778 Carefully read all of section 19 up to this point. Evaluate each one of
3779 the examples in the sections you read in the MathPiper worksheet you
3780 created or in the MathPiper console and verify that the results match the
3781 ones in the book. Copy all of the console examples you evaluated into your
3782 worksheet so they will be saved.