

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.....	8
1.1	Dedication.....	8
1.2	Acknowledgments.....	8
1.3	Support Email List.....	8
1.4	Recommended Weekly Sequence When Teaching A Class With This Book. .	8
2	Introduction.....	9
2.1	What Is A Mathematics Computing Environment?.....	9
2.2	What Is MathRider?.....	10
2.3	What Inspired The Creation Of Mathrider?.....	11
3	Downloading And Installing MathRider.....	13
3.1	Installing Sun's Java Implementation.....	13
3.1.1	Installing Java On A Windows PC.....	13
3.1.2	Installing Java On A Macintosh.....	13
3.1.3	Installing Java On A Linux PC.....	13
3.2	Downloading And Extracting.....	13
3.2.1	Extracting The Archive File For Windows Users.....	14
3.2.2	Extracting The Archive File For Unix Users.....	14
3.3	MathRider's Directory Structure & Execution Instructions.....	15
3.3.1	Executing MathRider On Windows Systems.....	15
3.3.2	Executing MathRider On Unix Systems.....	16
3.3.2.1	MacOS X.....	16
4	The Graphical User Interface.....	17
4.1	Buffers And Text Areas.....	17
4.2	The Gutter.....	17
4.3	Menus.....	17
4.3.1	File.....	18
4.3.2	Edit.....	18
4.3.3	Search.....	18
4.3.4	Markers, Folding, and View.....	19
4.3.5	Utilities.....	19
4.3.6	Macros.....	19
4.3.7	Plugins.....	19
4.3.8	Help.....	19
4.4	The Toolbar.....	19
4.4.1	Undo And Redo.....	20
5	MathPiper: A Computer Algebra System For Beginners.....	21
5.1	Numeric Vs. Symbolic Computations.....	21

5.2 Using The MathPiper Console As A Numeric (Scientific) Calculator.....	22
5.2.1 Functions.....	23
5.2.1.1 The Sqrt() Square Root Function.....	23
5.2.1.2 The IsEven() Function.....	24
5.2.2 Accessing Previous Input And Results.....	24
5.3 Saving And Restoring A Console Session.....	25
5.3.1 Syntax Errors.....	25
5.4 Using The MathPiper Console As A Symbolic Calculator.....	26
5.4.1 Variables.....	26
5.4.1.1 Calculating With Unbound Variables.....	27
5.4.1.2 Variable And Function Names Are Case Sensitive.....	29
5.4.1.3 Using More Than One Variable.....	29
5.5 Exercises.....	30
5.5.1 Exercise 1.....	30
5.5.2 Exercise 2.....	30
5.5.3 Exercise 3.....	30
5.5.4 Exercise 4.....	30
5.5.5 Exercise 5.....	31
6 The MathPiper Documentation Plugin.....	32
6.1 Function List.....	32
6.2 Mini Web Browser Interface.....	32
6.3 Exercises.....	33
6.3.1 Exercise 1.....	33
6.3.2 Exercise 2.....	33
7 Using MathRider As A Programmer's Text Editor.....	34
7.1 Creating, Opening, Saving, And Closing Text Files.....	34
7.2 Editing Files.....	34
7.3 File Modes.....	34
7.4 Learning How To Type Properly Is An Excellent Investment Of Your Time.....	35
7.5 Exercises.....	35
7.5.1 Exercise 1.....	35
8 MathRider Worksheet Files.....	36
8.1 Code Folds.....	36
8.1.1 The title Attribute.....	37
8.2 Automatically Inserting Folds & Removing Unpreserved Folds.....	37
8.3 Exercises.....	38
8.3.1 Exercise 1.....	38
8.3.2 Exercise 2.....	38
8.3.3 Exercise 3.....	38
8.3.4 Exercise 4.....	38
9 MathPiper Programming Fundamentals.....	39

9.1 Values and Expressions.....	39
9.2 Operators.....	39
9.3 Operator Precedence.....	40
9.4 Changing The Order Of Operations In An Expression.....	41
9.5 Functions & Function Names.....	42
9.6 Functions That Produce Side Effects.....	43
9.6.1 Printing Related Functions: Echo(), Write(), And Newline().....	43
9.6.1.1 Echo().....	43
9.6.1.2 Echo Statements Are Useful For "Debugging" Programs.....	45
9.6.1.3 Write().....	46
9.6.1.4 NewLine().....	46
9.7 Expressions Are Separated By Semicolons.....	47
9.7.1 Placing More Than One Expression On A Line In A Fold.....	47
9.7.2 Placing Multiple Expressions In A Code Block.....	48
9.7.2.1 Automatic Bracket, Parentheses, And Brace Match Indicating....	49
9.8 Strings.....	50
9.8.1 The MathPiper Console and MathPiper Folds Can Access The Same Variables.....	50
9.8.2 Using Strings To Make Echo's Output Easier To Read.....	51
9.8.2.1 Combining Strings With The : Operator.....	51
9.8.2.2 WriteString().....	51
9.8.2.3 Nl().....	52
9.8.2.4 Space().....	52
9.8.3 Accessing The Individual Letters In A String.....	52
9.9 Comments.....	53
9.10 Exercises.....	54
9.10.1 Exercise 1.....	55
9.10.2 Exercise 2.....	55
9.10.3 Exercise 3.....	55
9.10.4 Exercise 4.....	55
9.10.5 Exercise 5.....	55
9.10.6 Exercise 6.....	56
9.10.7 Exercise 7.....	56
10 Rectangular Selection Mode And Text Area Splitting.....	57
10.1 Rectangular Selection Mode.....	57
10.2 Text area splitting.....	57
10.3 Exercises.....	57
10.3.1 Exercise 1.....	58
11 Working With Random Integers.....	59
11.1 Obtaining Random Integers With The RandomInteger() Function.....	59
11.2 Simulating The Rolling Of Dice.....	60

11.3 Exercises.....	61
11.3.1 Exercise 1.....	61
12 Making Decisions.....	62
12.1 Conditional Operators.....	62
12.2 Predicate Expressions.....	65
12.3 Exercises.....	65
12.3.1 Exercise 1.....	65
12.3.2 Exercise 2.....	66
12.3.3 Exercise 3.....	66
12.4 Making Decisions With The If() Function & Predicate Expressions.....	66
12.4.1 If() Functions Which Include An "Else" Parameter.....	68
12.5 Exercises.....	68
12.5.1 Exercise 1.....	69
12.5.2 Exercise 2.....	69
12.6 The And(), Or(), & Not() Boolean Functions & Infix Notation.....	69
12.6.1 And().....	69
12.6.2 Or().....	71
12.6.3 Not() & Prefix Notation.....	72
12.7 Exercises.....	73
12.7.1 Exercise 1.....	74
12.7.2 Exercise 2.....	74
12.7.3 Exercise 3.....	74
13 The While() Looping Function & Bodied Notation.....	76
13.1 Printing The Integers From 1 to 10.....	76
13.2 Printing The Integers From 1 to 100.....	78
13.3 Printing The Odd Integers From 1 To 99.....	78
13.4 Printing The Integers From 1 To 100 In Reverse Order.....	79
13.5 Expressions Inside Of Code Blocks Are Indented.....	80
13.6 Long-Running Loops, Infinite Loops, & Interrupting Execution.....	80
13.7 A Program That Simulates Rolling Two Dice 50 Times.....	81
13.8 Exercises.....	83
13.8.1 Exercise 1.....	84
13.8.2 Exercise 2.....	84
13.8.3 Exercise 3.....	84
13.8.4 Exercise 4.....	84
14 Predicate Functions.....	85
14.1 Finding Prime Numbers With A Loop.....	86
14.2 Finding The Length Of A String With The Length() Function.....	88
14.3 Converting Numbers To Strings With The String() Function.....	89
14.4 Finding Prime Numbers Which End With 7 (And Multi-line Function Calls)	89

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

16.1.13 RemoveDuplicates()	110
16.1.14 Reverse()	110
16.1.15 Partition()	110
16.1.16 Table()	111
16.1.17 HeapSort()	112
16.2 Functions That Work With Integers	112
16.2.1 RandomIntegerVector()	112
16.2.2 Max() & Min()	112
16.2.3 Div() & Mod()	113
16.2.4 Gcd()	114
16.2.5 Lcm()	114
16.2.6 Sum()	115
16.2.7 Product()	115
16.3 Exercises	115
16.3.1 Exercise 1	116
16.3.2 Exercise 2	116
16.3.3 Exercise 3	116
16.3.4 Exercise 4	116
16.3.5 Exercise 5	116
16.3.6 Exercise 6	116
17 Nested Loops	117
17.1 Generate All The Combinations That Can Be Entered Into A Two Digit Wheel Lock Using A Nested Loop	117
17.2 Exercises	118
17.2.1 Exercise 1	119
17.2.2 Exercise 2	119
18 User Defined Functions	120
18.1 Global Variables, Local Variables, & Local()	122
18.2 Exercises	124
18.2.1 Exercise 1	124
18.2.2 Exercise 2	124
18.2.3 Exercise 3	124
19 Miscellaneous topics	125
19.1 Incrementing And Decrementing Variables With The ++ And -- Operators	125
19.1.1 Incrementing Variables With The ++ Operator	125
19.1.2 Decrementing Variables With The -- Operator	126
19.2 Exercises	127
19.2.1 Exercise 1	127

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:

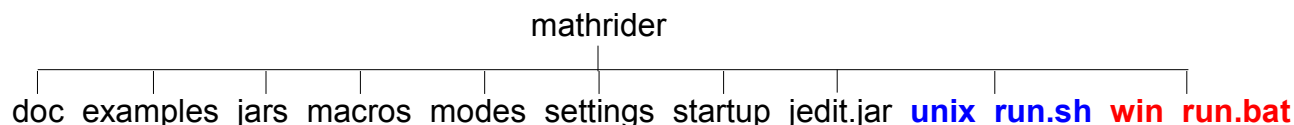


Illustration 1: MathRider's Directory Structure

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 e)

679 In> x + z

680 f)

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 category each function is contained in.
737 **Don't** include the **Alphabetical** or **Built In** categories in your search. For
738 example, the `N()` function is in the **Numbers (Operations)** category.

739 **7 Using MathRider As A Programmer's Text Editor**

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

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

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

753 **7.1 Creating, Opening, Saving, And Closing Text Files**

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

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

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

766 **7.2 Editing Files**

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

772 **7.3 File Modes**

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

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

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

782 ***7.4 Learning How To Type Properly Is An Excellent Investment Of Your*** 783 ***Time***

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

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

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

794 ***7.5 Exercises***

795 ***7.5.1 Exercise 1***

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

800 **8 MathRider Worksheet Files**

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

808 **8.1 Code Folds**

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

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

```
819 %mathpiper
820 2 + 3;
821 %/mathpiper
```

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

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

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

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

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

839 8.1.1 The title Attribute

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

```
843 %mathpiper,title="Add two numbers together."
```

```
844 2 + 3;
```

```
845 %/mathpiper
```

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

850 8.2 Automatically Inserting Folds & Removing Unpreserved Folds

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

```
852 %mathpiper
```

```
853 %/mathpiper
```

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

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

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

864 **8.3 Exercises**

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

867 [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)
868 [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)

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

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

876 **8.3.1 Exercise 1**

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

880 **8.3.2 Exercise 2**

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

883 **8.3.3 Exercise 3**

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

885 **8.3.4 Exercise 4**

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

889 a) $2 * 7 + 3$

890 b) $18 / 3$

891 c) $234238342 + 2038408203$

892 d) $324802984 * 2308098234$

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

894 9 MathPiper Programming Fundamentals

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

899 9.1 Values and Expressions

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

902 3

903 represents the number three, the value:

904 0.5

905 represents the number one half, and the value:

906 "Mathematics is powerful!"

907 represents an English sentence.

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

911 3

912 2 + 3

913 5 + 6*21/18 - 2^3

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

917 In> 2 + 3

918 Result> 5

919 9.2 Operators

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

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

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

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

930 In> 5 - 2
931 Result> 3

932 In> 3*4
933 Result> 12

934 In> 30/3
935 Result> 10

936 In> 8%5
937 Result> 3

938 In> 2^3
939 Result> 8

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

941 In> -3
942 Result> -3

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

945 In> - -3
946 Result> 3

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

952 **9.3 Operator Precedence**

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

960 [^] Exponents are evaluated right to left.

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

962 left to right.

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

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

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

966 5 + 6*21/18 - 2^3

967 And here it is in traditional form:

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

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

969 evaluates the operations in this expression:

970 5 + 6*21/18 - 2^3

971 5 + 6*21/18 - 8

972 5 + 126/18 - 8

973 5 + 7 - 8

974 12 - 8

975 4

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

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

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

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

980 **9.4 Changing The Order Of Operations In An Expression**

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

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

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

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

985 default precedence rules:

986 In> 2 + 4*5

987 Result> 22

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

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

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

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

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

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

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

- 1000 () Parentheses are evaluated from the inside out.
- 1001 ^ Then exponents are evaluated right to left.
- 1002 *,%,/ Then multiplication, remainder, and division operations are evaluated
1003 left to right.
- 1004 +, - Finally, addition and subtraction are evaluated left to right.

1005 **9.5 Functions & Function Names**

1006 In programming, **functions** are named blocks of code that can be executed one
1007 or more times by being **called** from other parts of the same program or called
1008 from other programs. Functions **can have values passed to them** from the
1009 calling code and they **always return a value** back to the calling code when they
1010 are finished executing. An example of a function is the **IsEven()** function which
1011 was discussed in an previous section.

1012 Functions are one way that MathPiper enables code to be reused. Most
1013 programming languages allow code to be reused in this way, although in other
1014 languages these named blocks of code are sometimes called **subroutines**,
1015 **procedures**, or **methods**.

1016 The functions that come with MathPiper have names which consist of either a
1017 single word (such as **Sum()**) or multiple words that have been put together to
1018 form a compound word (such as **IsBound()**). All letters in the names of
1019 functions which come with MathPiper are lower case except the beginning letter
1020 in each word, which are upper case.

1021 **9.6 Functions That Produce Side Effects**

1022 Most functions are executed to obtain the **results** they produce but some
1023 functions are executed in order to **have them perform work that is not in the**
1024 **form of a result**. Functions that perform work that is not in the form of a result
1025 are said to produce **side effects**. Side effects include many forms of work such
1026 as sending information to the user, opening files, and changing values in the
1027 computer's memory.

1028 When a function produces a side effect which sends information to the user, this
1029 information has the words **Side Effects:** placed before it in the output instead of
1030 the word **Result:**. The **Echo()** and **Write()** functions are examples of functions
1031 that produce side effects and they are covered in the next section.

1032 **9.6.1 Printing Related Functions: Echo(), Write(), And Newline()**

1033 The printing related functions send text information to the user and this is
1034 usually referred to as "printing" in this document. However, it may also be called
1035 "echoing" and "writing".

1036 **9.6.1.1 Echo()**

1037 The **Echo()** function takes one expression (or multiple expressions separated by
1038 commas) evaluates each expression, and then prints the results as side effect
1039 output. The following examples illustrate this:

```
1040 In> Echo(1)
1041 Result> True
1042 Side Effects>
1043 1
```

1044 In this example, the number 1 was passed to the Echo() function, the number
1045 was evaluated (all numbers evaluate to themselves), and the result of the
1046 evaluation was then printed as a side effect. Notice that Echo() **also returned a**
1047 **result**. In MathPiper, all functions return a result, but functions whose main
1048 purpose is to produce a side effect usually just return a result of **True** if the side
1049 effect succeeded or **False** if it failed. In this case, Echo() returned a result of
1050 **True** because it was able to successfully print a 1 as its side effect.

1051 The next example shows multiple expressions being sent to Echo() (notice that
1052 the expressions are separated by commas):

```
1053 In> Echo(1,1+2,2*3)
1054 Result> True
1055 Side Effects>
1056 1 3 6
```

1057 The expressions were each evaluated and their results were returned (separated
1058 by spaces) as side effect output. If it is desired that commas be printed between
1059 the numbers in the output, simply place three commas between the expressions
1060 that are passed to Echo():

```
1061 In> Echo(1,,,1+2,,,2*3)
1062 Result> True
1063 Side Effects>
1064 1 , 3 , 6
```

1065 Each time an Echo() function is executed, it always forces the display to drop
1066 down to the next line after it is finished. This can be seen in the following
1067 program which is similar to the previous one except it uses a separate Echo()
1068 function to display each expression:

```
1069 %mathpiper
1070 Echo(1);
1071 Echo(1+2);
1072 Echo(2*3);
1073 %/mathpiper
1074 %output,preserve="false"
1075 Result: True
1076
1077 Side Effects:
1078 1
1079 3
1080 6
1081 . %/output
```

1082 Notice how the 1, the 3, and the 6 are each on their own line.

1083 Now that we have seen how Echo() works, lets use it to do something useful. If
1084 more than one expression is evaluated in a %mathpiper fold, only the result from
1085 the last expression that was evaluated (which is usually the bottommost
1086 expression) is displayed:

```
1087 %mathpiper
1088 a := 1;
1089 b := 2;
1090 c := 3;
1091 %/mathpiper
```

```
1092     %output,preserve="false"
1093     Result: 3
1094 .    %/output
```

1095 In MathPiper, programs are executed one line at a time, starting at the topmost
1096 line of code and working downwards from there. In this example, the line `a := 1;`
1097 is executed first, then the line `b := 2;` is executed, and so on. Notice, however,
1098 that even though we wanted to see what was in all three variables, only the
1099 content of the last variable was displayed.

1100 The following example shows how `Echo()` can be used to display the contents of
1101 all three variables:

```
1102 %mathpiper
1103 a := 1;
1104 Echo(a);
1105 b := 2;
1106 Echo(b);
1107 c := 3;
1108 Echo(c);
1109 %/mathpiper
1110     %output,preserve="false"
1111     Result: True
1112
1113     Side Effects:
1114     1
1115     2
1116     3
1117 .    %/output
```

1118 9.6.1.2 Echo Statements Are Useful For "Debugging" Programs

1119 The errors that are in a program are often called "bugs". This name came from
1120 the days when computers were the size of large rooms and were made using
1121 electromechanical parts. Periodically, bugs would crawl into the machines and
1122 interfere with its moving mechanical parts and this would cause the machine to
1123 malfunction. The bugs needed to be located and removed before the machine
1124 would run properly again.

1125 Of course, even back then most program errors were produced by programmers
1126 entering wrong programs or entering programs wrong, but they liked to say that
1127 all of the errors were caused by bugs and not by themselves! The process of
1128 fixing errors in a program became known as **debugging** and the names "bugs"

1129 and "debugging" are still used by programmers today.

1130 One of the standard ways to locate bugs in a program is to place **Echo()** function
1131 calls in the code at strategic places which **print the contents of variables and**
1132 **display messages**. These Echo() functions will enable you to see what your
1133 program is doing while it is running. After you have found and fixed the bugs in
1134 your program, you can remove the debugging Echo() function calls or comment
1135 them out if you think they may be needed later.

1136 9.6.1.3 Write()

1137 The **Write()** function is similar to the Echo() function except it does not
1138 automatically drop the display down to the next line after it finishes executing:

```
1139 %mathpiper
1140 Write(1);
1141 Write(1+2);
1142 Echo(2*3);
1143 %/mathpiper
1144     %output,preserve="false"
1145     Result: True
1146
1147     Side Effects:
1148     1 3 6
1149 .    %/output
```

1150 Write() and Echo() have other differences besides the one discussed here and
1151 more information about them can be found in the documentation for these
1152 functions.

1153 9.6.1.4 NewLine()

1154 The **NewLine()** function simply prints a blank line in the side effects output. It
1155 is useful for placing vertical space between printed lines:

```
1156 %mathpiper
1157 a := 1;
1158 Echo(a);
1159 NewLine();
1160 b := 2;
1161 Echo(b);
```

```
1162 NewLine();
1163 c := 3;
1164 Echo(c);
1165 %/mathpiper
1166 %output,preserve="false"
1167 Result: True
1168
1169 Side Effects:
1170 1
1171 2
1172 3
1173 . %/output
```

1174 9.7 Expressions Are Separated By Semicolons

1175 As discussed earlier, all of the expressions that are inside of a **%mathpiper** fold
1176 must have a semicolon (;) after them. However, the expressions executed in the
1177 **MathPiper console** did not have a semicolon after them. MathPiper actually
1178 requires that all expressions end with a semicolon, but one does not need to add
1179 a semicolon to an expression which is typed into the MathPiper console **because**
1180 **the console adds it automatically** when the expression is executed.

1181 9.7.1 Placing More Than One Expression On A Line In A Fold

1182 All the previous code examples have had each of their expressions on a separate
1183 line, but multiple expressions can also be placed on a single line because the
1184 semicolons tell MathPiper where one expression ends and the next one begins:

```
1185 %mathpiper
1186 a := 1; Echo(a); b := 2; Echo(b); c := 3; Echo(c);
1187 %/mathpiper
1188 %output,preserve="false"
1189 Result: True
1190
1191 Side Effects:
1192 1
1193 2
1194 3
1195 . %/output
```

1196 The spaces that are in the code of this example are used to make the code more
1197 readable. Any spaces that are present within any expressions or between them
1198 are ignored by MathPiper and if we remove the spaces from the previous code,
1199 the output remains the same:

```
1200 %mathpiper
1201 a:=1;Echo(a);b:=2;Echo(b);c:= 3;Echo(c);
1202 %/mathpiper
1203     %output,preserve="false"
1204     Result: True
1205
1206     Side Effects:
1207     1
1208     2
1209     3
1210 .    %/output
```

1211 9.7.2 Placing Multiple Expressions In A Code Block

1212 A **code block** (which is also called a **compound expression**) consists of one or
1213 more expressions which are separated by semicolons and placed within an open
1214 bracket (**[**) and close bracket (**]**) pair. When a code block is evaluated, each
1215 expression in the block will be executed from left to right. The following
1216 example shows expressions being executed within of a code block inside the
1217 MathPiper console:

```
1218 In> [a := 1; Echo(a); b := 2; Echo(b); c := 3; Echo(c);]
1219 Result> True
1220 Side Effects>
1221 1
1222 2
1223 3
```

1224 Notice that all of the expressions were executed and 1-3 was printed as a side
1225 effect. Code blocks **always return the result of the last expression executed**
1226 **as the result of the whole block**. In this case, **True** was returned as the result
1227 because the last **Echo(c)** function returned **True**. If we place **another**
1228 **expression after the Echo(c) function**, however, **the block will execute this**
1229 **new expression last and its result will be the one returned by the block**:

```
1230 In> [a := 1; Echo(a); b := 2; Echo(b); c := 3; Echo(c); 2 + 2;]
1231 Result> 4
1232 Side Effects>
1233 1
```


1234 2
1235 3

1236 Finally, code blocks can have their contents placed on separate lines if desired:

```
1237 %mathpiper
1238 [
1239     a := 1;
1240     Echo(a);
1241     b := 2;
1242     Echo(b);
1243     c := 3;
1244     Echo(c);
1245 ];
1251 %/mathpiper
1252     %output,preserve="false"
1253     Result: True
1254     Side Effects:
1255     1
1256     2
1257     3
1258     . %/output
```

1260 Code blocks are very powerful and we will be discussing them further in later
1261 sections.

1262 9.7.2.1 Automatic Bracket, Parentheses, And Brace Match Indicating

1263 In programming, most open brackets '[' have a close bracket ']', most open
1264 parentheses '(' have a close parentheses ')', and most open braces '{' have a
1265 close brace '}'. It is often difficult to make sure that each "open" character has a
1266 matching "close" character and if any of these characters don't have a match,
1267 then an error will be produced.

1268 Thankfully, most programming text editors have a character match indicating
1269 tool that will help locate problems. To try this tool, paste the following code into
1270 a .mrw file and following the directions that are present in its comments:

```
1271 %mathpiper
1272 /*
```

```
1273 Copy this code into a .mrw file. Then, place the cursor
1274 to the immediate right of any {, }, [, ], (, or ) character.
1275 You should notice that the match to this character is
1276 indicated by a rectangle being drawing around it.
1277 */
```

```
1278 list := {1,2,3};
1279 [
1280     Echo("Hello");
1281     Echo(list);
1282 ];
1283 %/mathpiper
```

1284 9.8 Strings

1285 A **string** is a **value** that is used to hold text-based information. The typical
1286 expression that is used to create a string consists of **text which is enclosed**
1287 **within double quotes**. Strings can be assigned to variables just like numbers
1288 can and strings can also be displayed using the Echo() function. The following
1289 program assigns a string value to the variable 'a' and then echos it to the user:

```
1290 %mathpiper
1291 a := "Hello, I am a string.";
1292 Echo(a);
1293 %/mathpiper
1294 %output,preserve="false"
1295 Result: True
1296
1297 Side Effects:
1298 Hello, I am a string.
1299 . %/output
```

1300 9.8.1 The MathPiper Console and MathPiper Folds Can Access The Same 1301 Variables

1302 A useful aspect of using MathPiper inside of MathRider is that variables that are
1303 assigned inside of a **%mathpiper fold** are accessible inside of the **MathPiper**
1304 **console** and variables that are assigned inside of the **MathPiper console** are
1305 available inside of **%mathpiper folds**. For example, after the above fold is
1306 executed, the string that has been bound to variable 'a' can be displayed in the
1307 MathPiper console:

```
1308 In> a
1309 Result> "Hello, I am a string."
```

1310 9.8.2 Using Strings To Make Echo's Output Easier To Read

1311 When the Echo() function is used to print the values of multiple variables, it is
1312 often helpful to print some information next to each variable so that it is easier to
1313 determine which value came from which Echo() function in the code. The
1314 following program prints the name of the variable that each value came from
1315 next to it in the side effects output:

```
1316 %mathpiper
1317 a := 1;
1318 Echo("Variable a: ", a);
1319 b := 2;
1320 Echo("Variable b: ", b);
1321 c := 3;
1322 Echo("Variable c: ", c);
1323 %/mathpiper
1324     %output,preserve="false"
1325     Result: True
1326
1327     Side Effects:
1328     Variable a: 1
1329     Variable b: 2
1330     Variable c: 3
1331 .    %/output
```

1332 9.8.2.1 Combining Strings With The : Operator

1333 If you need to combine two or more strings into one string, you can use the :
1334 operator like this:

```
1335 In> "A" : "B" : "C"
1336 Result: "ABC"
1337
1337 In> "Hello " : "there!"
1338 Result: "Hello there!"
```

1339 9.8.2.2 WriteString()

1340 The **WriteString()** function prints a string without shows the double quotes that

1341 are around it.. For example, here is the Write() function being used to print the
1342 string "Hello":

```
1343 In> Write("Hello")
1344 Result: True
1345 Side Effects:
1346 "Hello"
```

1347 Notice the double quotes? Here is how the WriteString() function prints "Hello":

```
1348 In> WriteString("Hello")
1349 Result: True
1350 Side Effects:
1351 Hello
```

1352 **9.8.2.3 NI()**

1353 The **NI()** (New Line) function is used with the : function to place newline
1354 characters inside of strings:

```
1355 In> WriteString("A": NI() : "B")
1356 Result: True
1357 Side Effects:
1358 A
1359 B
```

1360 **9.8.2.4 Space()**

1361 The Space() function is used to add spaces to printed output:

```
1362 In> WriteString("A"); Space(5); WriteString("B")
1363 Result: True
1364 Side Effects:
1365 A      B
```

```
1366 In> WriteString("A"); Space(10); WriteString("B")
1367 Result: True
1368 Side Effects:
1369 A          B
```

```
1370 In> WriteString("A"); Space(20); WriteString("B")
1371 Result: True
1372 Side Effects:
1373 A                      B
```

1374 **9.8.3 Accessing The Individual Letters In A String**

1375 Individual letters in a string (which are also called **characters**) can be accessed
1376 by placing the character's position number (also called an **index**) inside of

1377 brackets **[]** after the variable it is bound to. A character's position is determined
1378 by its distance from the left side of the string starting at 1. For example, in the
1379 string "Hello", 'H' is at position 1, 'e' is at position 2, etc. The following code
1380 shows individual characters in the above string being accessed:

```
1381 In> a := "Hello, I am a string."  
1382 Result> "Hello, I am a string."
```

```
1383 In> a[1]  
1384 Result> "H"
```

```
1385 In> a[2]  
1386 Result> "e"
```

```
1387 In> a[3]  
1388 Result> "l"
```

```
1389 In> a[4]  
1390 Result> "l"
```

```
1391 In> a[5]  
1392 Result> "o"
```

1393 **9.9 Comments**

1394 Source code can often be difficult to understand and therefore all programming
1395 languages provide the ability for **comments** to be included in the code.

1396 Comments are used to explain what the code near them is doing and they are
1397 usually meant to be read by humans instead of being processed by a computer.
1398 Therefore, comments are ignored by the computer when a program is executed.

1399 There are two ways that MathPiper allows comments to be added to source code.
1400 The first way is by placing two forward slashes **//** to the left of any text that is
1401 meant to serve as a comment. The text from the slashes to the end of the line
1402 the slashes are on will be treated as a comment. Here is a program that contains
1403 comments which use slashes:

```
1404 %mathpiper  
1405 //This is a comment.
```

```
1406 x := 2; //Set the variable x equal to 2.
```

```
1407 %/mathpiper
```

```
1408     %output,preserve="false"  
1409     Result: 2  
1410 .    %/output
```

1411 When this program is executed, any text that starts with slashes is ignored.
1412 The second way to add comments to a MathPiper program is by enclosing the
1413 comments inside of slash-asterisk/asterisk-slash symbols `/* */`. This option is
1414 useful when a comment is too large to fit on one line. Any text between these
1415 symbols is ignored by the computer. This program shows a longer comment
1416 which has been placed between these symbols:

```
1417 %mathpiper
1418 /*
1419  This is a longer comment and it uses
1420  more than one line. The following
1421  code assigns the number 3 to variable
1422  x and then returns it as a result.
1423 */
1424 x := 3;
1425 %/mathpiper
1426     %output,preserve="false"
1427     Result: 3
1428 .    %/output
```

1429 9.10 Exercises

1430 For the following exercises, create a new MathRider worksheet file called
1431 **book_1_section_9_exercises_<your first name>_<your last name>.mrw**.
1432 (**Note: there are no spaces in this file name**). For example, John Smith's
1433 worksheet would be called:

1434 **book_1_section_9_exercises_john_smith.mrw**.

1435 After this worksheet has been created, place your answer for each exercise that
1436 requires a fold into its own fold in this worksheet. Place a title attribute in the
1437 start tag of each fold which indicates the exercise the fold contains the solution
1438 to. The folds you create should look similar to this one:

```
1439 %mathpiper,title="Exercise 1"
1440 //Sample fold.
1441 %/mathpiper
```

1442 If an exercise uses the MathPiper console instead of a fold, copy the work you
1443 did in the console into the worksheet so it can be saved.

1444 9.10.1 Exercise 1

1445 Carefully read all of section 9. Evaluate each one of the examples in
1446 section 9 in the MathPiper worksheet you created or in the MathPiper
1447 console and verify that the results match the ones in the book. Copy all
1448 of the console examples you evaluated into your worksheet so they will be
1449 saved.

1450 9.10.2 Exercise 2

1451 Change the precedence of the following expression using parentheses so that
1452 it prints 20 instead of 14:

1453 $2 + 3 * 4$

1454 9.10.3 Exercise 3

1455 Place the following calculations into a fold and then use one Echo()
1456 function per variable to print the results of the calculations. Put
1457 strings in the Echo() functions which indicate which variable each
1458 calculated value is bound to:

1459 $a := 1+2+3+4+5;$
1460 $b := 1-2-3-4-5;$
1461 $c := 1*2*3*4*5;$
1462 $d := 1/2/3/4/5;$

1463 9.10.4 Exercise 4

1464 Place the following calculations into a fold and then use one Echo()
1465 function to print the results of all the calculations on a single line
1466 (Remember, the Echo() function can print multiple values if they are
1467 separated by commas.):

1468 `Clear(x);`
1469 $a := 2*2*2*2*2;$
1470 $b := 2^5;$
1471 $c := x^2 * x^3;$
1472 $d := 2^2 * 2^3;$

1473 9.10.5 Exercise 5

1474 The following code assigns a string which contains all of the upper case
1475 letters of the alphabet to the variable **upper**. Each of the three Echo()
1476 functions prints an index number and the letter that is at that position in
1477 the string. Place this code into a fold and then continue the Echo()
1478 functions so that all 26 letters and their index numbers are printed

1479 `upper := "ABCDEFGHIJKLMNOPQRSTUVWXYZ";`
1480 `Echo(1,upper[1]);`
1481 `Echo(2,upper[2]);`

```
1482 Echo(3,upper[3]);
```

1483 9.10.6 Exercise 6

```
1484 Use Echo() functions to print an index number and the character at this
1485 position for the following string (this is similar to what was done in
1486 Exercise 4.):
```

```
1487 extra := "!.@#$$%^&*() _+<>,?/{ }[]|\-= ";
```

```
1488 Echo(1,extra[1]);
```

```
1489 Echo(2,extra[2]);
```

```
1490 Echo(3,extra[3]);
```

1491 9.10.7 Exercise 7

```
1492 The following program uses strings and index numbers to print a person's
1493 name. Create a program which uses the three strings from this program to
1494 print the names of three of your favorite movie actors.
```

```
1495 %mathpiper
```

```
1496 /*
```

```
1497 This program uses strings and index numbers to print
```

```
1498 a person's name.
```

```
1499 */
```

```
1500 upper := "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

```
1501 lower := "abcdefghijklmnopqrstuvwxyz";
```

```
1502 extra := "!.@#$$%^&*() _+<>,?/{ }[]|\-= ";
```

```
1503 //Print "Mary Smith."
```

```
1504 Echo(upper[13],lower[1],lower[18],lower[25],extra[12],upper[19],lower[13],l
```

```
1505 ower[9],lower[20],lower[8],extra[1]);
```

```
1506 %/mathpiper
```

```
1507 %output,preserve="false"
```

```
1508 Result: True
```

```
1509
```

```
1510 Side Effects:
```

```
1511 Mary Smith.
```

```
1512 . %/output
```


1513 10 Rectangular Selection Mode And Text Area Splitting

1514 10.1 Rectangular Selection Mode

1515 One capability that MathRider has that a word processor may not have is the
1516 ability to select rectangular sections of text. To see how this works, do the
1517 following:

- 1518 1) Type three or four lines of text into a text area.
- 1519 2) Hold down the **<Alt>** key then slowly press the **backslash key** (\) a few
1520 times. The bottom of the MathRider window contains a text field which
1521 MathRider uses to communicate information to the user. As **<Alt>** is
1522 repeatedly pressed, messages are displayed which read **Rectangular**
1523 **selection is on** and **Rectangular selection is off**.
- 1524 3) Turn rectangular selection on and then select some text in order to see
1525 how this is different than normal selection mode. **When you are done**
1526 **experimenting, set rectangular selection mode to off.**

1527 Most of the time normal selection mode is what you want to use but in certain
1528 situations rectangular selection mode is better.

1529 10.2 Text area splitting

1530 Sometimes it is useful to have two or more text areas open for a single document
1531 or multiple documents so that different parts of the documents can be edited at
1532 the same time. A situation where this would have been helpful was in the
1533 previous section where the output from an exercise in a MathRider worksheet
1534 contained a list of index numbers and letters which was useful for completing a
1535 later exercise.

1536 MathRider has this ability and it is called **splitting**. If you look just to the right
1537 of the toolbar there is an icon which looks like a blank window, an icon to the
1538 right of it which looks like a window which was split horizontally, and an icon to
1539 the right of the horizontal one which is split vertically. If you let your mouse
1540 hover over these icons, a short description will be displayed for each of them.

1541 Select a text area and then experiment with splitting it by pressing the horizontal
1542 and vertical splitting buttons. Move around these split text areas with their
1543 scroll bars and when you want to unsplit the document, just press the "**Unsplit**
1544 **All**" icon.

1545 10.3 Exercises

1546 For the following exercises, create a new MathRider worksheet file called
1547 **book_1_section_10_exercises_<your first name>_<your last name>.mrw**.

1548 **(Note: there are no spaces in this file name)**. For example, John Smith's
1549 worksheet would be called:

1550 **book_1_section_10_exercises_john_smith.mrw.**

1551 For the following exercises, simply type your answers anywhere in the
1552 worksheet.

1553 **10.3.1 Exercise 1**

1554 Carefully read all of section 9 then answer the following questions:

1555 a) Give two examples where rectangular selection mode may be more useful
1556 than regular selection mode.

1557 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
```

```
1598 In> RandomInteger(100)
1599 Result> 82
1600 In> RandomInteger(100)
1601 Result> 93
1602 In> RandomInteger(100)
1603 Result> 32
```

1604 A range of random integers that does not start with 1 can also be generated by
1605 using the **two argument** version of **RandomInteger()**. For example, random
1606 integers between 25 and 75 can be obtained by passing RandomInteger() the
1607 lowest integer in the range and the highest one:

```
1608 In> RandomInteger(25, 75)
1609 Result: 28
1610 In> RandomInteger(25, 75)
1611 Result: 37
1612 In> RandomInteger(25, 75)
1613 Result: 58
1614 In> RandomInteger(25, 75)
1615 Result: 50
1616 In> RandomInteger(25, 75)
1617 Result: 70
```

1618 **11.2 Simulating The Rolling Of Dice**

1619 The following example shows the simulated rolling of a single six sided die using
1620 the RandomInteger() function:

```
1621 In> RandomInteger(6)
1622 Result> 5
1623 In> RandomInteger(6)
1624 Result> 6
1625 In> RandomInteger(6)
1626 Result> 3
1627 In> RandomInteger(6)
1628 Result> 2
1629 In> RandomInteger(6)
1630 Result> 5
```

1631 Code that simulates the rolling of two 6 sided dice can be evaluated in the
1632 MathPiper console by placing it within a **code block**. The following code
1633 outputs the sum of the two simulated dice:

```
1634 In> [a := RandomInteger(6); b := RandomInteger(6); a + b;]
1635 Result> 6
1636 In> [a := RandomInteger(6); b := RandomInteger(6); a + b;]
1637 Result> 12
1638 In> [a := RandomInteger(6); b := RandomInteger(6); a + b;]
1639 Result> 6
```

```
1640 In> [a := RandomInteger(6); b := RandomInteger(6); a + b;]
1641 Result> 4
1642 In> [a := RandomInteger(6); b := RandomInteger(6); a + b;]
1643 Result> 3
1644 In> [a := RandomInteger(6); b := RandomInteger(6); a + b;]
1645 Result> 8
```

1646 Now that we have the ability to simulate the rolling of two 6 sided dice, it would
1647 be interesting to determine if some sums of these dice occur more frequently
1648 than other sums. What we would like to do is to roll these simulated dice
1649 hundreds (or even thousands) of times and then analyze the sums that were
1650 produced. We don't have the programming capability to easily do this yet, but
1651 after we finish the section on **while loops**, we will.

1652 11.3 Exercises

1653 For the following exercises, create a new MathRider worksheet file called
1654 **book_1_section_11_exercises_<your first name>_<your last name>.mrw.**
1655 (**Note: there are no spaces in this file name**). For example, John Smith's
1656 worksheet would be called:

1657 **book_1_section_11_exercises_john_smith.mrw.**

1658 After this worksheet has been created, place your answer for each exercise that
1659 requires a fold into its own fold in this worksheet. Place a title attribute in the
1660 start tag of each fold which indicates the exercise the fold contains the solution
1661 to. The folds you create should look similar to this one:

```
1662 %mathpiper,title="Exercise 1"
1663 //Sample fold.
1664 %/mathpiper
```

1665 If an exercise uses the MathPiper console instead of a fold, copy the work you
1666 did in the console into the worksheet so it can be saved.

1667 11.3.1 Exercise 1

1668 Carefully read all of section 11. Evaluate each one of the examples in
1669 section 11 in the MathPiper worksheet you created or in the MathPiper
1670 console and verify that the results match the ones in the book. Copy all
1671 of the console examples you evaluated into your worksheet so they will be
1672 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
```

```
1693 In> 4 > 5
1694 Result> False
```

```
1695 In> 8 >= 8
1696 Result> True
```

```
1697 In> 5 <= 10
1698 Result> True
```

1699 The following examples show each of the conditional operators in Table 2 being
1700 used to compare values that have been assigned to variables **x** and **y**:

```
1701 %mathpiper
```

```
1702 // Example 1.
```

```
1703 x := 2;
```

```
1704 y := 3;
```

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

```
1711 %/mathpiper
```

```
1712 %output,preserve="false"
```

```
1713 Result: True
```

```
1714
```

```
1715 Side Effects:
```

```
1716 2 = 3 :False
```

```
1717 2 != 3 :True
```

```
1718 2 < 3 :True
```

```
1719 2 <= 3 :True
```

```
1720 2 > 3 :False
```

```
1721 2 >= 3 :False
```

```
1722 . %/output
```

```
1723 %mathpiper
```

```
1724 // Example 2.
```

```
1725 x := 2;
```

```
1726 y := 2;
```

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

```
1733 %/mathpiper
```

```
1734     %output,preserve="false"
```

```
1735     Result: True
```

```
1736
```

```
1737     Side Effects:
```

```
1738     2 = 2 :True
```

```
1739     2 != 2 :False
```

```
1740     2 < 2 :False
```

```
1741     2 <= 2 :True
```

```
1742     2 > 2 :False
```

```
1743     2 >= 2 :True
```

```
1744 .    %/output
```

```
1745 %mathpiper
```

```
1746 // Example 3.
```

```
1747 x := 3;
```

```
1748 y := 2;
```

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

```
1755 %/mathpiper
```

```
1756     %output,preserve="false"
```

```
1757     Result: True
```

```
1758
```

```
1759     Side Effects:
```

```
1760     3 = 2 :False
```

```
1761     3 != 2 :True
```

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

```
1763     3 <= 2 :False
```

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

```
1765     3 >= 2 :True
```

```
1766 .    %/output
```

1767 Conditional operators are placed at a lower level of precedence than the other
1768 operators we have covered to this point:

1769 () Parentheses are evaluated from the inside out.

1770 ^ Then exponents are evaluated right to left.

1771 *,%,/ Then multiplication, remainder, and division operations are evaluated
1772 left to right.

1773 +, - Then addition and subtraction are evaluated left to right.

1774 =,!=,<,<=,>,>= Finally, conditional operators are evaluated.

1775 **12.2 Predicate Expressions**

1776 Expressions which return either **True** or **False** are called "**predicate**"
1777 expressions. By themselves, predicate expressions are not very useful and they
1778 only become so when they are used with special decision making functions, like
1779 the If() function (which is discussed in the next section).

1780 **12.3 Exercises**

1781 For the following exercises, create a new MathRider worksheet file called
1782 **book_1_section_12a_exercises_<your first name>_<your last name>.mrw.**
1783 (**Note: there are no spaces in this file name**). For example, John Smith's
1784 worksheet would be called:

1785 **book_1_section_12a_exercises_john_smith.mrw.**

1786 After this worksheet has been created, place your answer for each exercise that
1787 requires a fold into its own fold in this worksheet. Place a title attribute in the
1788 start tag of each fold which indicates the exercise the fold contains the solution
1789 to. The folds you create should look similar to this one:

1790 `%mathpiper,title="Exercise 1"`

1791 `//Sample fold.`

1792 `%/mathpiper`

1793 If an exercise uses the MathPiper console instead of a fold, copy the work you
1794 did in the console into the worksheet so it can be saved.

1795 **12.3.1 Exercise 1**

1796 Carefully read all of section 12 up to this point. Evaluate each one of
1797 the examples in the sections you read in the MathPiper worksheet you
1798 created or in the MathPiper console and verify that the results match the
1799 ones in the book. Copy all of the console examples you evaluated into your
1800 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

1827 that accepts a predicate expression as a parameter can also accept the boolean
1828 values True and False).

1829 The following program uses an **If()** function to determine if the value in variable
1830 number is greater than 5. If number is greater than 5, the program will echo
1831 "Greater" and then "End of program":

```
1832 %mathpiper
1833 number := 6;
1834 If(number > 5, Echo(number, "is greater than 5.));
1835 Echo("End of program.");
1836 %/mathpiper
1837 %output,preserve="false"
1838 Result: True
1839
1840 Side Effects:
1841 6 is greater than 5.
1842 End of program.
1843 . %/output
```

1844 In this program, number has been set to 6 and therefore the expression number
1845 > 5 is **True**. When the **If()** function evaluates the **predicate expression** and
1846 determines it is **True**, it then executes the **first Echo()** function. The **second**
1847 **Echo()** function at the bottom of the program prints "End of program"
1848 regardless of what the If() function does. (**Note: semicolons cannot be placed**
1849 **after expressions which are in function calls.**)

1850 Here is the same program except that **number** has been set to **4** instead of **6**:

```
1851 %mathpiper
1852 number := 4;
1853 If(number > 5, Echo(number, "is greater than 5.));
1854 Echo("End of program.");
1855 %/mathpiper
1856 %output,preserve="false"
1857 Result: True
1858
1859 Side Effects:
1860 End of program.
1861 . %/output
```

1862 This time the expression **number > 4** returns a value of **False** which causes the
1863 **If()** function to not execute the "**then**" expression that was passed to it.

1864 12.4.1 If() Functions Which Include An "Else" Parameter

1865 The second form of the If() function takes a third "**else**" expression which is
1866 executed only if the predicate expression is **False**. This program is similar to the
1867 previous one except an "**else**" expression has been added to it:

```
1868 %mathpiper
1869 x := 4;
1870 If(x > 5, Echo(x, "is greater than 5."), Echo(x, "is NOT greater than 5.));
1871 Echo("End of program.");
1872 %/mathpiper
1873     %output, preserve="false"
1874     Result: True
1875
1876     Side Effects:
1877     4 is NOT greater than 5.
1878     End of program.
1879 .    %/output
```

1880 12.5 Exercises

1881 For the following exercises, create a new MathRider worksheet file called
1882 **book_1_section_12b_exercises_<your first name>_<your last name>.mrw**.
1883 (**Note: there are no spaces in this file name**). For example, John Smith's
1884 worksheet would be called:

1885 **book_1_section_12b_exercises_john_smith.mrw**.

1886 After this worksheet has been created, place your answer for each exercise that
1887 requires a fold into its own fold in this worksheet. Place a title attribute in the
1888 start tag of each fold which indicates the exercise the fold contains the solution
1889 to. The folds you create should look similar to this one:

```
1890 %mathpiper, title="Exercise 1"
1891 //Sample fold.
1892 %/mathpiper
```

1893 If an exercise uses the MathPiper console instead of a fold, copy the work you

1894 did in the console into the worksheet so it can be saved.

1895 **12.5.1 Exercise 1**

1896 Carefully read all of section 12 starting at the end of the previous
1897 exercises and up to this point. Evaluate each one of the examples in the
1898 sections you read in the MathPiper worksheet you created or in the
1899 MathPiper console and verify that the results match the ones in the book.
1900 Copy all of the console examples you evaluated into your worksheet so they
1901 will be saved.

1902 **12.5.2 Exercise 2**

1903 Write a program which uses the RandomInteger() function to simulate the
1904 flipping of a coin (Hint: you can use 1 to represent a head and 0 to
1905 represent a tail.). Use predicate expressions, the If() function, and the
1906 Echo() function to print the string "**The coin came up heads.**" or the string
1907 "**The coin came up tails.**", depending on what the simulated coin flip came
1908 up as when the code was executed.

1909 **12.6 The And(), Or(), & Not() Boolean Functions & Infix Notation**

1910 **12.6.1 And()**

1911 Sometimes a programmer needs to check if two or more expressions are all **True**
1912 and one way to do this is with the **And()** function. The And() function has **two**
1913 **calling formats** (or **notations**) and this is the first one:

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

1914 This calling format is able to accept one or more predicate expressions as input.
1915 If **all** of these expressions returns a value of **True**, the And() function will also
1916 return a **True**. However, if **any** of the expressions return a **False**, then the And()
1917 function will return a **False**. This can be seen in the following example:

```
1918 In> And(True, True)  
1919 Result> True
```

```
1920 In> And(True, False)  
1921 Result> False
```

```
1922 In> And(False, True)  
1923 Result> False
```

```
1924 In> And(True, True, True, True)  
1925 Result> True
```

```
1926 In> And(True, True, False, True)
1927 Result> False
```

1928 The second format (or notation) that can be used to call the And() function is
1929 called **infix** notation:

```
expression1 And expression2
```

1930 With **infix** notation, an expression is placed on both sides of the And() function
1931 name instead of being placed inside of parentheses that are next to it:

```
1932 In> True And True
1933 Result> True
```

```
1934 In> True And False
1935 Result> False
```

```
1936 In> False And True
1937 Result> False
```

1938 Infix notation can only accept **two** expressions at a time, but it is often more
1939 convenient to use than function calling notation. The following program also
1940 demonstrates the infix version of the And() function being used:

```
1941 %mathpiper
```

```
1942 a := 7;
1943 b := 9;
```

```
1944 Echo("1: ", a < 5 And b < 10);
1945 Echo("2: ", a > 5 And b > 10);
1946 Echo("3: ", a < 5 And b > 10);
1947 Echo("4: ", a > 5 And b < 10);
```

```
1948 If(a > 5 And b < 10, Echo("These expressions are both true."));
```

```
1949 %/mathpiper
```

```
1950     %output,preserve="false"
1951     Result: True
1952
1953     Side Effects:
1954     1: False
1955     2: False
1956     3: False
1957     4: True
1958     These expressions are both true.
1959 .    %/output
```

1960 **12.6.2 Or()**

1961 The Or() function is similar to the And() function in that it has both a function
1962 calling format and an infix calling format and it only works with predicate
1963 expressions. However, instead of requiring that all expressions be **True** in order
1964 to return a **True**, Or() will return a **True** if **one or more expressions are True**.

1965 Here is the function calling format for Or():

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

1966 and this example shows Or() being used with function calling format:

1967 In> Or(True, False)

1968 Result> True

1969 In> Or(False, True)

1970 Result> True

1971 In> Or(False, False)

1972 Result> False

1973 In> Or(False, False, False, False)

1974 Result> False

1975 In> Or(False, True, False, False)

1976 Result> True

1977 The infix notation format for Or() is as follows:

```
expression1 Or expression2
```

1978 and this example shows infix notation being used:

1979 In> True Or False

1980 Result> True

1981 In> False Or True

1982 Result> True

1983 In> False Or False

1984 Result> False

1985 The following program also demonstrates the infix version of the Or() function
1986 being used:

```
1987 %mathpiper
1988 a := 7;
1989 b := 9;
1990 Echo("1: ", a < 5 Or b < 10);
1991 Echo("2: ", a > 5 Or b > 10);
1992 Echo("3: ", a > 5 Or b < 10);
1993 Echo("4: ", a < 5 Or b > 10);
1994 If(a < 5 Or b < 10, Echo("At least one of these expressions is true."));
1995 %/mathpiper
1996 %output,preserve="false"
1997 Result: True
1998
1999 Side Effects:
2000 1: True
2001 2: True
2002 3: True
2003 4: False
2004 At least one of these expressions is true.
2005 . %/output
```

2006 12.6.3 Not() & Prefix Notation

2007 The **Not()** function works with predicate expressions like the And() and Or()
2008 functions do, except it can only accept **one** expression as input. The way Not()
2009 works is that it changes a **True** value to a **False** value and a **False** value to a
2010 **True** value. Here is the Not()'s function calling format:

```
Not(expression)
```

2011 and this example shows Not() being used with function calling format:

```
2012 In> Not(True)
2013 Result> False
2014 In> Not(False)
2015 Result> True
```

2016 Instead of providing an alternative infix calling format like And() and Or() do,
2017 Not()'s second calling format uses **prefix** notation:

```
Not expression
```


2018 Prefix notation looks similar to function notation except no parentheses are used:

```
2019 In> Not True
2020 Result> False
```

```
2021 In> Not False
2022 Result> True
```

2023 Finally, here is a program that also uses the prefix version of Not():

```
2024 %mathpiper
2025 Echo("3 = 3 is ", 3 = 3);
2026 Echo("Not 3 = 3 is ", Not 3 = 3);
2027 %/mathpiper
2028     %output,preserve="false"
2029     Result: True
2030
2031     Side Effects:
2032     3 = 3 is True
2033     Not 3 = 3 is False
2034 .    %/output
```

2035 12.7 Exercises

2036 For the following exercises, create a new MathRider worksheet file called
2037 **book_1_section_12c_exercises_<your first name>_<your last name>.mrw**.
2038 (**Note: there are no spaces in this file name**). For example, John Smith's
2039 worksheet would be called:

2040 **book_1_section_12c_exercises_john_smith.mrw**.

2041 After this worksheet has been created, place your answer for each exercise that
2042 requires a fold into its own fold in this worksheet. Place a title attribute in the
2043 start tag of each fold which indicates the exercise the fold contains the solution
2044 to. The folds you create should look similar to this one:

```
2045 %mathpiper,title="Exercise 1"
2046 //Sample fold.
2047 %/mathpiper
```

2048 If an exercise uses the MathPiper console instead of a fold, copy the work you

2049 did in the console into the worksheet so it can be saved.

2050 **12.7.1 Exercise 1**

2051 Carefully read all of section 12 starting at the end of the previous
2052 exercises and up to this point. Evaluate each one of the examples in the
2053 sections you read in the MathPiper worksheet you created or in the
2054 MathPiper console and verify that the results match the ones in the book.
2055 Copy all of the console examples you evaluated into your worksheet so they
2056 will be saved.

2057 **12.7.2 Exercise 2**

2058 The following program simulates the rolling of two dice and prints a
2059 message if **both** of the two dice come up less than or equal to 3. Create a
2060 similar program which simulates the flipping of two coins and print the
2061 message "Both coins came up heads." if both coins come up heads.

```
2062 %mathpiper
2063 /*
2064    This program simulates the rolling of two dice and prints a message if
2065    both of the two dice come up less than or equal to 3.
2066 */
```

```
2067 dice1 := RandomInteger(6);
2068 dice2 := RandomInteger(6);
```

```
2069 Echo("Dice1: ", dice1, "   Dice2: ", dice2);
2070 NewLine();
```

```
2071 If( dice1 <= 3 And dice2 <= 3, Echo("Both dice came up <= to 3.") );
```

```
2072 %/mathpiper
```

2073 **12.7.3 Exercise 3**

2074 The following program simulates the rolling of two dice and prints a
2075 message if **either** of the two dice come up less than or equal to 3. Create
2076 a similar program which simulates the flipping of two coins and print the
2077 message "At least one coin came up heads." if at least one coin comes up
2078 heads.

```
2079 %mathpiper
2080 /*
2081    This program simulates the rolling of two dice and prints a message if
2082    either of the two dice come up less than or equal to 3.
2083 */
```

```
2084 dice1 := RandomInteger(6);
2085 dice2 := RandomInteger(6);
```

```
2086 Echo("Dice1: ", dice1, "   Dice2: ", dice2);
2087 NewLine();

2088 If( dice1 <= 3 Or dice2 <= 3, Echo("At least one die came up <= 3.") );

2089 %/mathpiper
```

2090 13 The While() Looping Function & Bodied Notation

2091 Many kinds of machines, including computers, derive much of their power from
2092 the principle of **repeated cycling**. **Repeated cycling** in a MathPiper program
2093 means to execute one or more expressions over and over again and this process
2094 is called "**looping**". MathPiper provides a number of ways to implement **loops**
2095 in a program and these ways range from straight-forward to subtle.

2096 We will begin discussing looping in MathPiper by starting with the straight-
2097 forward **While** function. The calling format for the **While** function is as follows:

```
2098 While(predicate)  
2099 [  
2100     body_expressions  
2101 ];
```

2102 The **While** function is similar to the **If** function except it will repeatedly execute
2103 the expressions it contains as long as its "predicate" expression is **True**. As soon
2104 as the predicate expression returns a **False**, the While() function skips the
2105 expressions it contains and execution continues with the expression that
2106 immediately follows the While() function (if there is one).

2107 The expressions which are contained in a While() function are called its "**body**"
2108 and all functions which have body expressions are called "**bodied**" functions. If
2109 a body contains more than one expression then these expressions need to be
2110 placed within a **code block** (code blocks were discussed in an earlier section).
2111 What a function's body is will become clearer after studying some example
2112 programs.

2113 13.1 Printing The Integers From 1 to 10

2114 The following program uses a While() function to print the integers from 1 to 10:

```
2115 %mathpiper  
  
2116 // This program prints the integers from 1 to 10.  
  
2117 /*  
2118     Initialize the variable count to 1  
2119     outside of the While "loop".  
2120 */  
2121 count := 1;  
  
2122 While(count <= 10)  
2123 [  
2124     Echo(count);
```

```
2125
2126     count := count + 1; //Increment count by 1.
2127 1;

2128 %/mathpiper

2129     %output,preserve="false"
2130     Result: True
2131
2132     Side Effects:
2133     1
2134     2
2135     3
2136     4
2137     5
2138     6
2139     7
2140     8
2141     9
2142     10
2143 . %/output
```

2144 In this program, a single variable called **count** is created. It is used to tell the
2145 Echo() function which integer to print and it is also used in the predicate
2146 expression that determines if the While() function should continue to **loop** or not.

2147 When the program is executed, 1 is placed into **count** and then the While()
2148 function is called. The predicate expression **count** <= 10 becomes **1** <= 10
2149 and, since 1 is indeed less than or equal to 10, a value of **True** is returned by the
2150 predicate expression.

2151 The While() function sees that the predicate expression returned a **True** and
2152 therefore it executes all of the expressions inside of its **body** from top to bottom.

2153 The Echo() function prints the current contents of count (which is 1) and then the
2154 expression count := count + 1 is executed.

2155 The expression **count := count + 1** is a standard expression form that is used in
2156 many programming languages. Each time an expression in this form is
2157 evaluated, it **increases the variable it contains by 1**. Another way to describe
2158 the effect this expression has on **count** is to say that it **increments count by 1**.

2159 In this case **count** contains **1** and, after the expression is evaluated, **count**
2160 contains **2**.

2161 After the last expression inside the body of the While() function is executed, the
2162 While() function reevaluates its predicate expression to determine whether it
2163 should continue looping or not. Since **count** is **2** at this point, the predicate
2164 expression returns **True** and the code inside the body of the While() function is
2165 executed again. This loop will be repeated until **count** is incremented to **11** and
2166 the predicate expression returns **False**.

2167 **13.2 Printing The Integers From 1 to 100**

2168 The previous program can be adjusted in a number of ways to achieve different
2169 results. For example, the following program prints the integers from 1 to 100 by
2170 changing the **10** in the predicate expression to **100**. A Write() function is used in
2171 this program so that its output is displayed on the same line until it encounters
2172 the **wrap margin** in MathRider (which can be set in Utilities -> Buffer
2173 Options...).

```
2174 %mathpiper
2175 // Print the integers from 1 to 100.
2176 count := 1;
2177 While(count <= 100)
2178 [
2179     Write(count,,);
2180
2181     count := count + 1; //Increment count by 1.
2182 ];
2183 %/mathpiper
2184     %output,preserve="false"
2185     Result: True
2186
2187     Side Effects:
2188     1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,
2189     24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,
2190     44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,
2191     64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,
2192     84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100
2193 . %/output
```

2194 **13.3 Printing The Odd Integers From 1 To 99**

2195 The following program prints the odd integers from 1 to 99 by changing the
2196 **increment value** in the increment expression from **1** to **2**:

```
2197 %mathpiper
2198 //Print the odd integers from 1 to 99.
2199 x := 1;
2200 While(x <= 100)
2201 [
2202     Write(x,,);
```

```
2203     x := x + 2;    //Increment x by 2.
2204 ];
2205 %/mathpiper
2206     %output,preserve="false"
2207     Result: True
2208
2209     Side Effects:
2210     1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39,41,43,
2211     45,47,49,51,53,55,57,59,61,63,65,67,69,71,73,75,77,79,81,83,
2212     85,87,89,91,93,95,97,99
2213 .    %/output
```

2214 **13.4 Printing The Integers From 1 To 100 In Reverse Order**

2215 Finally, the following program prints the integers from 1 to 100 in reverse order:

```
2216 %mathpiper
2217 //Print the integers from 1 to 100 in reverse order.
2218 x := 100;
2219 While(x >= 1)
2220 [
2221     Write(x,,);
2222     x := x - 1;    //Decrement x by 1.
2223 ];
2224 %/mathpiper
2225     %output,preserve="false"
2226     Result: True
2227
2228     Side Effects:
2229     100,99,98,97,96,95,94,93,92,91,90,89,88,87,86,85,84,83,82,
2230     81,80,79,78,77,76,75,74,73,72,71,70,69,68,67,66,65,64,63,
2231     62,61,60,59,58,57,56,55,54,53,52,51,50,49,48,47,46,45,44,
2232     43,42,41,40,39,38,37,36,35,34,33,32,31,30,29,28,27,26,25,
2233     24,23,22,21,20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,
2234     3,2,1
2235 .    %/output
```

2236 In order to achieve the reverse ordering, this program had to initialize **x** to **100**,
2237 check to see if **x** was **greater than or equal to 1** ($x \geq 1$), and **decrement** **x** by
2238 **subtracting 1 from it** instead of adding 1 to it.

2239 **13.5 Expressions Inside Of Code Blocks Are Indented**

2240 In the programs in the previous sections which use while loops, notice that the
2241 expressions which are inside of the While() function's code block are **indented**.
2242 These expressions do not need to be indented to execute properly, but doing so
2243 makes the program easier to read.

2244 **13.6 Long-Running Loops, Infinite Loops, & Interrupting Execution**

2245 It is easy to create a loop that will execute a **large number of times**, or even **an**
2246 **infinite number of times**, either on purpose or by mistake. When you execute
2247 a program that contains an **infinite loop**, it will run until you tell MathPiper to
2248 **interrupt** its execution. This is done by opening the MathPiper console and then
2249 pressing the "**Stop**" button which it contains. The Stop button is circular and it
2250 has an X on it. (**Note: currently this button only works if MathPiper is**
2251 **executed inside of a %mathpiper fold.**)

2252 Lets experiment with the **Stop** button by executing a program that contains an
2253 infinite loop and then stopping it:

```
2254 %mathpiper
2255 //Infinite loop example program.
2256 x := 1;
2257 While(x < 10)
2258 [
2259     x := 3; //Oops, x is not being incremented!.
2260 ];
2261 %/mathpiper
2262     %output,preserve="false"
2263     Processing...
2264 . %/output
```

2265 Since the contents of x is never changed inside the loop, the expression **x < 10**
2266 always evaluates to **True** which causes the loop to continue looping. Notice that
2267 the %output fold contains the word "**Processing...**" to indicate that the program
2268 is still running the code.

2269 Execute this program now and then interrupt it using the "**Stop**" button. When
2270 the program is interrupted, the %output fold will display the message "**User**
2271 **interrupted calculation**" to indicate that the program was interrupted. After a
2272 program has been interrupted, the program can be edited and then rerun.

2273 **13.7 A Program That Simulates Rolling Two Dice 50 Times**

2274 The following program is larger than the previous programs that have been
2275 discussed in this book, but it is also more interesting and more useful. It uses a
2276 While() loop to simulate the rolling of two dice 50 times and it records how many
2277 times each possible sum has been rolled so that this data can be printed. The
2278 comments in the code explain what each part of the program does. (Remember, if
2279 you copy this program to a MathRider worksheet, you can use **rectangular**
2280 **selection mode** to easily remove the line numbers).

```
2281 %mathpiper
2282 /*
2283     This program simulates rolling two dice 50 times.
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 */
2290 numberOfTwosRolled := 0;
2291 numberOfThreesRolled := 0;
2292 numberOfFoursRolled := 0;
2293 numberOfFivesRolled := 0;
2294 numberOfSixesRolled := 0;
2295 numberOfSevensRolled := 0;
2296 numberOfEightsRolled := 0;
2297 numberOfNinesRolled := 0;
2298 numberOfTensRolled := 0;
2299 numberOfElevensRolled := 0;
2300 numberOfTwelvesRolled := 0;

2301 //This variable keeps track of the number of the current roll.
2302 roll := 1;

2303 Echo("These are the rolls:");

2304 /*
2305     The simulation is performed inside of this while loop. The number of
2306     times the dice will be rolled can be changed by changing the number 50
2307     which is in the While function's predicate expression.
2308 */
2309 While(roll <= 50)
2310 [
2311     //Roll the dice.
2312     die1 := RandomInteger(6);
2313     die2 := RandomInteger(6);
```

```
2314
2315
2316 //Calculate the sum of the two dice.
2317 rollSum := die1 + die2;
2318
2319
2320 /*
2321  Print the sum that was rolled.  Note: if a large number of rolls
2322  is going to be performed (say > 1000), it would be best to comment
2323  out this Write() function so that it does not put too much text
2324  into the output fold.
2325 */
2326 Write(rollSum,,);
2327
2328
2329 /*
2330  These If() functions determine which sum was rolled and then add
2331  1 to the variable which is keeping track of the number of times
2332  that sum was rolled.
2333 */
2334 If(rollSum = 2, numberOfTwosRolled := numberOfTwosRolled + 1);
2335 If(rollSum = 3, numberOfThreesRolled := numberOfThreesRolled + 1);
2336 If(rollSum = 4, numberOfFoursRolled := numberOfFoursRolled + 1);
2337 If(rollSum = 5, numberOfFivesRolled := numberOfFivesRolled + 1);
2338 If(rollSum = 6, numberOfSixesRolled := numberOfSixesRolled + 1);
2339 If(rollSum = 7, numberOfSevensRolled := numberOfSevensRolled + 1);
2340 If(rollSum = 8, numberOfEightsRolled := numberOfEightsRolled + 1);
2341 If(rollSum = 9, numberOfNinesRolled := numberOfNinesRolled + 1);
2342 If(rollSum = 10, numberOfTensRolled := numberOfTensRolled + 1);
2343 If(rollSum = 11, numberOfElevensRolled := numberOfElevensRolled+1);
2344 If(rollSum = 12, numberOfTwelvesRolled := numberOfTwelvesRolled+1);
2345
2346
2347 //Increment the roll variable to the next roll number.
2348 roll := roll + 1;
2349 ];

```



```
2350 //Print the contents of the sum count variables for visual analysis.
2351 NewLine();
2352 NewLine();
2353 Echo("Number of Twos rolled: ", numberOfTwosRolled);
2354 Echo("Number of Threes rolled: ", numberOfThreesRolled);
2355 Echo("Number of Fours rolled: ", numberOfFoursRolled);
2356 Echo("Number of Fives rolled: ", numberOfFivesRolled);
2357 Echo("Number of Sixes rolled: ", numberOfSixesRolled);
2358 Echo("Number of Sevens rolled: ", numberOfSevensRolled);
2359 Echo("Number of Eights rolled: ", numberOfEightsRolled);
2360 Echo("Number of Nines rolled: ", numberOfNinesRolled);
2361 Echo("Number of Tens rolled: ", numberOfTensRolled);
2362 Echo("Number of Elevens rolled: ", numberOfElevensRolled);
2363 Echo("Number of Twelves rolled: ", numberOfTwelvesRolled);
```

```
2364 %/mathpiper
2365     %output,preserve="false"
2366     Result: True
2367
2368     Side effects:
2369     These are the rolls:
2370     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,
2371     12,7,8,12,6,8,10,10,5,9,8,4,5,3,5,7,7,4,6,7,7,5,8,
2372
2373     Number of Twos rolled: 0
2374     Number of Threes rolled: 3
2375     Number of Fours rolled: 6
2376     Number of Fives rolled: 4
2377     Number of Sixes rolled: 6
2378     Number of Sevens rolled: 13
2379     Number of Eights rolled: 6
2380     Number of Nines rolled: 3
2381     Number of Tens rolled: 2
2382     Number of Elevens rolled: 4
2383     Number of Twelves rolled: 3
2384 .    %/output
```

2385 13.8 Exercises

2386 For the following exercises, create a new MathRider worksheet file called
2387 **book_1_section_13_exercises_<your first name>_<your last name>.mrw.**
2388 **(Note: there are no spaces in this file name).** For example, John Smith's
2389 worksheet would be called:

2390 **book_1_section_13_exercises_john_smith.mrw.**

2391 After this worksheet has been created, place your answer for each exercise that
2392 requires a fold into its own fold in this worksheet. Place a title attribute in the
2393 start tag of each fold which indicates the exercise the fold contains the solution
2394 to. The folds you create should look similar to this one:

```
2395 %mathpiper,title="Exercise 1"
```

```
2396 //Sample fold.
```

```
2397 %/mathpiper
```

2398 If an exercise uses the MathPiper console instead of a fold, copy the work you
2399 did in the console into the worksheet so it can be saved.

2400 13.8.1 Exercise 1

2401 Carefully read all of section 13 up to this point. Evaluate each one of
2402 the examples in the sections you read in the MathPiper worksheet you
2403 created or in the MathPiper console and verify that the results match the
2404 ones in the book. Copy all of the console examples you evaluated into your
2405 worksheet so they will be saved.

2406 13.8.2 Exercise 2

2407 Create a program which uses a while loop to print the even integers from 2
2408 to 50 inclusive.

2409 13.8.3 Exercise 3

2410 Create a program which prints all the multiples of 5 between 5 and 50
2411 inclusive.

2412 13.8.4 Exercise 4

2413 Create a program which simulates the flipping of a coin 500 times. Print
2414 the number of times the coin came up heads and the number of times it came
2415 up tails after the loop is finished executing.

2416 14 Predicate Functions

2417 A **predicate function** is a function that either returns **True** or **False**. Most
2418 predicate functions in MathPiper have names which begin with "**Is**". For
2419 example, **IsEven()**, **IsOdd()**, **IsInteger()**, etc. The following examples show
2420 some of the predicate functions that are in MathPiper:

```
2421 In> IsEven(4)
2422 Result> True
```

```
2423 In> IsEven(5)
2424 Result> False
```

```
2425 In> IsZero(0)
2426 Result> True
```

```
2427 In> IsZero(1)
2428 Result> False
```

```
2429 In> IsNegativeInteger(-1)
2430 Result> True
```

```
2431 In> IsNegativeInteger(1)
2432 Result> False
```

```
2433 In> IsPrime(7)
2434 Result> True
```

```
2435 In> IsPrime(100)
2436 Result> False
```

2437 There is also an **IsBound()** and an **IsUnbound()** function that can be used to
2438 determine whether or not a value is bound to a given variable:

```
2439 In> a
2440 Result> a
```

```
2441 In> IsBound(a)
2442 Result> False
```

```
2443 In> a := 1
2444 Result> 1
```

```
2445 In> IsBound(a)
2446 Result> True
```

```
2447 In> Clear(a)
2448 Result> True
```

```
2449 In> a
2450 Result> a
```

```
2451 In> IsBound(a)
2452 Result> False
```

2453 The complete list of predicate functions is contained in the **User**
2454 **Functions/Predicates** node in the MathPiperDocs plugin.

2455 **14.1 Finding Prime Numbers With A Loop**

2456 Predicate functions are very powerful when they are combined with loops
2457 because they can be used to automatically make numerous checks. The
2458 following program uses a while loop to pass the integers 1 through 20 (one at a
2459 time) to the **IsPrime()** function in order to determine which integers are prime
2460 and which integers are not prime:

```
2461 %mathpiper
2462 //Determine which numbers between 1 and 20 (inclusive) are prime.
2463 x := 1;
2464 While(x <= 20)
2465 [
2466     primeStatus := IsPrime(x);
2467     Echo(x, "is prime: ", primeStatus);
2468     x := x + 1;
2469 ];
2470
2471 %/mathpiper
2472
2473 %output,preserve="false"
2474 Result: True
2475
2476 Side Effects:
2477 1 is prime: False
2478 2 is prime: True
2479 3 is prime: True
2480 4 is prime: False
2481 5 is prime: True
2482 6 is prime: False
2483 7 is prime: True
2484 8 is prime: False
2485 9 is prime: False
2486 10 is prime: False
2487 11 is prime: True
2488 12 is prime: False
```

```
2489         13 is prime: True
2490         14 is prime: False
2491         15 is prime: False
2492         16 is prime: False
2493         17 is prime: True
2494         18 is prime: False
2495         19 is prime: True
2496         20 is prime: False
2497     .    %/output
```

2498 This program worked fairly well, but it is limited because it prints a line for each
2499 prime number and also each non-prime number. This means that if large ranges
2500 of integers were processed, enormous amounts of output would be produced.
2501 The following program solves this problem by using an If() function to only print
2502 a number if it is prime:

```
2503 %mathpiper
2504 //Print the prime numbers between 1 and 50 (inclusive).
2505 x := 1;
2506 While(x <= 50)
2507 [
2508     primeStatus := IsPrime(x);
2509
2510     If(primeStatus = True, Write(x,,) );
2511
2512     x := x + 1;
2513 ];
2514 %/mathpiper
2515     %output,preserve="false"
2516     Result: True
2517
2518     Side Effects:
2519     2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,
2520 .    %/output
```

2521 This program is able to process a much larger range of numbers than the
2522 previous one without having its output fill up the text area. However, the
2523 program itself can be shortened by moving the **IsPrime()** function **inside** of the
2524 **If()** function instead of using the **primeStatus** variable to communicate with it:

```
2525 %mathpiper
2526 /*
```

```
2527     Print the prime numbers between 1 and 50 (inclusive).
2528     This is a shorter version which places the IsPrime() function
2529     inside of the If() function instead of using a variable.
2530 */
2531 x := 1;
2532 While(x <= 50)
2533 [
2534     If(IsPrime(x), Write(x,,) );
2535
2536     x := x + 1;
2537 ];
2538 %/mathpiper
2539     %output,preserve="false"
2540     Result: True
2541
2542     Side Effects:
2543     2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,
2544     . %/output
```

2545 **14.2 Finding The Length Of A String With The Length() Function**

2546 Strings can contain zero or more characters and the **Length()** function can be
2547 used to determine how many characters a string holds:

```
2548 In> s := "Red"
2549 Result> "Red"
```

```
2550 In> Length(s)
2551 Result> 3
```

2552 In this example, the string "Red" is assigned to the variable **s** and then **s** is
2553 passed to the **Length()** function. The **Length()** function returned a **3** which
2554 means the string contained **3 characters**.

2555 The following example shows that strings can also be passed to functions
2556 directly:

```
2557 In> Length("Red")
2558 Result> 3
```

2559 An **empty string** is represented by **two double quote marks with no space in**
2560 **between them**. The **length** of an empty string is **0**:


```
2561 In> Length("")
2562 Result> 0
```

2563 **14.3 Converting Numbers To Strings With The String() Function**

2564 Sometimes it is useful to convert a number to a string so that the individual
2565 digits in the number can be analyzed or manipulated. The following example
2566 shows a **number** being converted to a **string** with the **String()** function so that
2567 its **leftmost** and **rightmost** digits can be placed into **variables**:

```
2568 In> number := 523
2569 Result> 523
```

```
2570 In> stringNumber := String(number)
2571 Result> "523"
```

```
2572 In> leftmostDigit := stringNumber[1]
2573 Result> "5"
```

```
2574 In> rightmostDigit := stringNumber[ Length(stringNumber) ]
2575 Result> "3"
```

2576 Notice that the Length() function is used here to determine which character in
2577 **stringNumber** held the **rightmost** digit.

2578 **14.4 Finding Prime Numbers Which End With 7 (And Multi-line Function Calls)**

2580 Now that we have covered how to turn a number into a string, lets use this
2581 ability inside a loop. The following program finds all the **prime numbers**
2582 between **1** and **500** which have a **7 as their rightmost digit**. There are three
2583 important things which are shown in this program:

2584 1) Function calls **can have their parameters placed on more than one**
2585 **line** if the parameters are too long to fit on a **single line**. In this case, a long
2586 code block is being placed inside of an If() function.

2587 2) Code blocks (which are considered to be compound expressions) **cannot**
2588 **have a semicolon placed after them if they are in a function call**. If a
2589 semicolon is placed after this code block, an error will be produced.

2590 3) If() functions can be placed inside of other If() functions in order to make
2591 more complex decisions. This is referred to as **nesting** functions.

2592 When the program is executed, it finds 24 prime numbers which have 7 as their
2593 rightmost digit:

```
2594 %mathpiper
2595 /*
2596     Find all the prime numbers between 1 and 500 which have a 7
2597     as their rightmost digit.
2598 */
2599 x := 1;
2600 While(x <= 500)
2601 [
2602     //Notice how function parameters can be put on more than one line.
2603     If(IsPrime(x),
2604         [
2605             stringVersionOfNumber := String(x);
2606             stringLength := Length(stringVersionOfNumber);
2607             //Notice that If() functions can be placed inside of other
2608             // If() functions.
2609             If(stringVersionOfNumber[stringLength] = "7", Write(x,,) );
2610         ] //Notice that semicolons cannot be placed after code blocks
2611         //which are in function calls.
2612     ); //This is the close parentheses for the outer If() function.
2613     x := x + 1;
2614 ];
2615
2620 %/mathpiper
2621 %output,preserve="false"
2622     Result: True
2623
2624     Side Effects:
2625     7,17,37,47,67,97,107,127,137,157,167,197,227,257,277,307,317,
2626     337,347,367,397,457,467,487,
2627 . %/output
```

2628 It would be nice if we had the ability to store these numbers someplace so that
2629 they could be processed further and this is discussed in the next section.

2630 14.5 Exercises

2631 For the following exercises, create a new MathRider worksheet file called
2632 **book_1_section_14_exercises_<your first name>_<your last name>.mrw.**
2633 (**Note: there are no spaces in this file name**). For example, John Smith's
2634 worksheet would be called:

2635 book_1_section_14_exercises_john_smith.mrw.

2636 After this worksheet has been created, place your answer for each exercise that
2637 requires a fold into its own fold in this worksheet. Place a title attribute in the
2638 start tag of each fold which indicates the exercise the fold contains the solution
2639 to. The folds you create should look similar to this one:

```
2640 %mathpiper,title="Exercise 1"
```

```
2641 //Sample fold.
```

```
2642 %/mathpiper
```

2643 If an exercise uses the MathPiper console instead of a fold, copy the work you
2644 did in the console into the worksheet so it can be saved.

2645 14.5.1 Exercise 1

2646 Carefully read all of section 14 up to this point. Evaluate each one of
2647 the examples in the sections you read in the MathPiper worksheet you
2648 created or in the MathPiper console and verify that the results match the
2649 ones in the book. Copy all of the console examples you evaluated into your
2650 worksheet so they will be saved.

2651 14.5.2 Exercise 2

2652 Write a program which uses a loop to determine how many prime numbers there
2653 are between 1 and 1000. You do not need to print the numbers themselves,
2654 just how many there are.

2655 14.5.3 Exercise 3

2656 Write a program which uses a loop to print all of the prime numbers between
2657 10 and 99 which contain the digit 3 in either their 1's place, or their
2658 10's place, or both places.

2659 15 Lists: Values That Hold Sequences Of Expressions

2660 The **list** value type is designed to hold expressions in an **ordered collection** or
2661 **sequence**. Lists are very flexible and they are one of the most heavily used
2662 value types in MathPiper. Lists can **hold expressions of any type**, they can be
2663 made to **grow and shrink as needed**, and they can be **nested**. Expressions in a
2664 list can be **accessed by their position** in the list (similar to the way that
2665 characters in a string are accessed) and they can also be **replaced by other**
2666 **expressions**.

2667 One way to create a list is by placing zero or more expressions separated by
2668 commas inside of a **pair of braces {}**. In the following example, a list is created
2669 that contains various expressions and then it is assigned to the variable **x**:

```
2670 In> x := {7,42,"Hello",1/2,var}  
2671 Result> {7,42,"Hello",1/2,var}
```

```
2672 In> x  
2673 Result> {7,42,"Hello",1/2,var}
```

2674 The number of expressions in a list can be determined with the **Length()**
2675 function:

```
2676 In> Length({7,42,"Hello",1/2,var})  
2677 Result> 5
```

2678 A single expression in a list can be accessed by placing a set of **brackets []** to
2679 the right of the variable that is bound to the list and then putting the
2680 expression's position number inside of the brackets (**Note: the first expression**
2681 **in the list is at position 1 counting from the left end of the list**):

```
2682 In> x[1]  
2683 Result> 7
```

```
2684 In> x[2]  
2685 Result> 42
```

```
2686 In> x[3]  
2687 Result> "Hello"
```

```
2688 In> x[4]  
2689 Result> 1/2
```

```
2690 In> x[5]  
2691 Result> var
```

2692 The **1st** and **2nd** expressions in this list are **integers**, the **3rd** expression is a

2693 **string**, the **4th** expression is a **rational number** and the **5th** expression is an
2694 **unbound variable**.

2695 Lists can also hold other lists as shown in the following example:

```
2696 In> x := {20, 30, {31, 32, 33}, 40}
```

```
2697 Result> {20,30,{31,32,33},40}
```

```
2698 In> x[1]
```

```
2699 Result> 20
```

```
2700 In> x[2]
```

```
2701 Result> 30
```

```
2702 In> x[3]
```

```
2703 Result> {31,32,33}
```

```
2704 In> x[4]
```

```
2705 Result> 40
```

```
2706
```

2707 The expression in the **3rd** position in the list is another **list** which contains the
2708 integers **31**, **32**, and **33**.

2709 An expression in this second list can be accessed by two **two sets of brackets**:

```
2710 In> x[3][2]
```

```
2711 Result> 32
```

2712 The **3** inside of the first set of brackets accesses the **3rd** member of the **first** list
2713 and the **2** inside of the second set of brackets accesses the **2nd** member of the
2714 **second** list.

2715 **15.1 Append() & Nondestructive List Operations**

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

2716 The **Append()** function adds an expression to the end of a list:

```
2717 In> testList := {21,22,23}
```

```
2718 Result> {21,22,23}
```

```
2719 In> Append(testList, 24)
```

```
2720 Result> {21,22,23,24}
```

2721 However, instead of changing the **original** list, **Append()** creates a **copy** of the
2722 **original** list and appends the expression to the **copy**. This can be confirmed by
2723 evaluating the variable **testList** after the **Append()** function has been called:

```
2724 In> testList
2725 Result> {21,22,23}
```

2726 Notice that the list that is bound to **testList** was not modified by the **Append()**
2727 function. This is called a **nondestructive list operation** and **most MathPiper**
2728 **functions that manipulate lists do so nondestructively**. To have the new list
2729 bound to the variable that is being used, the following technique can be
2730 employed:

```
2731 In> testList := {21,22,23}
2732 Result> {21,22,23}

2733 In> testList := Append(testList, 24)
2734 Result> {21,22,23,24}

2735 In> testList
2736 Result> {21,22,23,24}
```

2737 After this code has been executed, the new list has indeed been bound to
2738 **testList** as desired.

2739 There are some functions, such as **DestructiveAppend()**, which **do** change the
2740 original list and most of them begin with the word "Destructive". These are
2741 called "destructive functions" and they are advanced functions which are not
2742 covered in this book.

2743 **15.2 Using While Loops With Lists**

2744 Functions that loop can be used to **select each expression in a list in turn** so
2745 that an operation can be performed on these expressions. The following
2746 program uses a while loop to print each of the expressions in a list:

```
2747 %mathpiper
2748 //Print each number in the list.

2749 x := {55,93,40,21,7,24,15,14,82};
2750 y := 1;

2751 While(y <= Length(x))
2752 [
2753     Echo(y, "- ", x[y]);
2754     y := y + 1;
2755 ];

2756 %/mathpiper

2757 %output,preserve="false"
```

```
2758         Result: True
2759
2760         Side Effects:
2761         1 - 55
2762         2 - 93
2763         3 - 40
2764         4 - 21
2765         5 - 7
2766         6 - 24
2767         7 - 15
2768         8 - 14
2769         9 - 82
2770     .    %/output
```

2771 A **loop** can also be used to search through a list. The following program uses a
2772 **While()** function and an **If()** function to search through a list to see if it contains
2773 the number **53**. If 53 is found in the list, a message is printed:

```
2774 %mathpiper

2775 //Determine if 53 is in the list.

2776 testList := {18,26,32,42,53,43,54,6,97,41};
2777 index := 1;

2778 While(index <= Length(testList))
2779 [
2780     If(testList[index] = 53,
2781         Echo("53 was found in the list at position", index));
2782     index := index + 1;
2783 ];

2785 %/mathpiper

2786 %output,preserve="false"
2787     Result: True
2788
2789     Side Effects:
2790     53 was found in the list at position 5
2791 .    %/output
```

2792 When this program was executed, it determined that **53** was present in the list at
2793 position **5**.

2794 15.2.1 Using A While Loop And Append() To Place Values In A List

2795 In an earlier section it was mentioned that it would be nice if we could store a set
2796 of values for later processing and this can be done with a **while loop** and the

2797 **Append()** function. The following program creates an empty list and assigned it
2798 to the variable **primes**. The **while loop** and the **IsPrime()** function is then used
2799 to locate the prime integers between 1 and 50 and the **Append()** function is used
2800 to place them in the list. The last part of the program then prints some
2801 information about the numbers that were placed into the list:

```
2802 %mathpiper
2803 //Place the prime numbers between 1 and 50 (inclusive) into a list.
2804 //Create an empty list.
2805 primes := {};
2806 x := 1;
2807 While(x <= 50)
2808 [
2809     /*
2810         If x is prime, append it to the end of the list and then assign
2811         the new list that is created to the variable 'primes'.
2812     */
2813     If(IsPrime(x), primes := Append(primes, x ) );
2814
2815     x := x + 1;
2816 ];
2817 //Print information about the primes that were found.
2818 Echo("Primes ", primes);
2819 Echo("The number of primes in the list = ", Length(primes) );
2820 Echo("The first number in the list = ", primes[1] );
2821 %/mathpiper
2822     %output,preserve="false"
2823     Result: True
2824
2825     Side Effects:
2826     Primes {2,3,5,7,11,13,17,19,23,29,31,37,41,43,47}
2827     The number of primes in the list = 15
2828     The first number in the list = 2
2829 .    %/output
```

2830 The ability to place values into a list with a loop is very powerful and we will be
2831 using this ability throughout the rest of the book.

2832 15.3 Exercises

2833 For the following exercises, create a new MathRider worksheet file called
2834 **book_1_section_15a_exercises_<your first name>_<your last name>.mrw.**

2835 **(Note: there are no spaces in this file name)**. For example, John Smith's
2836 worksheet would be called:

2837 **book_1_section_15a_exercises_john_smith.mrw.**

2838 After this worksheet has been created, place your answer for each exercise that
2839 requires a fold into its own fold in this worksheet. Place a title attribute in the
2840 start tag of each fold which indicates the exercise the fold contains the solution
2841 to. The folds you create should look similar to this one:

2842 `%mathpiper,title="Exercise 1"`

2843 `//Sample fold.`

2844 `%/mathpiper`

2845 If an exercise uses the MathPiper console instead of a fold, copy the work you
2846 did in the console into the worksheet so it can be saved.

2847 **15.3.1 Exercise 1**

2848 Carefully read all of section 15 up to this point. Evaluate each one of
2849 the examples in the sections you read in the MathPiper worksheet you
2850 created or in the MathPiper console and verify that the results match the
2851 ones in the book. Copy all of the console examples you evaluated into your
2852 worksheet so they will be saved.

2853 **15.3.2 Exercise 2**

2854 Create a program that uses a loop and an IsOdd() function to analyze the
2855 following list and then print the number of odd numbers it contains.

2856 `{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}`

2857 **15.3.3 Exercise 3**

2858 Create a program that uses a loop and an IsNegativeNumber() function to
2859 copy all of the negative numbers in the following list into a new list.
2860 Use the variable **negativeNumbers** to hold the new list.

2861 `{36,-29,-33,-6,14,7,-16,-3,-14,37,-38,-8,-45,-21,-26,6,6,38,-20,33,41,-`
2862 `4,24,37,40,29}`

2863 **15.3.4 Exercise 4**

2864 Create a program that uses a loop to analyze the following list and then
2865 print the following information about it:

- 2866 1) The largest number in the list.
2867 2) The smallest number in the list.
2868 3) The sum of all the numbers in the list.

```
2869 {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}
```

2870 **15.4 The ForEach() Looping Function**

2871 The **ForEach()** function uses a **loop** to index through a list like the While()
2872 function does, but it is more flexible and automatic. ForEach() also uses bodied
2873 notation like the While() function and here is its calling format:

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

2874 **ForEach()** selects each expression in a list in turn, assigns it to the passed-in
2875 "variable", and then executes the expressions that are inside of "body".
2876 Therefore, body is **executed once for each expression in the list**.

2877 **15.5 Print All The Values In A List Using A ForEach() function**

2878 This example shows how ForEach() can be used to print all of the items in a list:

```
2879 %mathpiper
```

```
2880 //Print all values in a list.
```

```
2881 ForEach(value, {50,51,52,53,54,55,56,57,58,59})
```

```
2882 [
```

```
2883     Echo(value);
```

```
2884 ];
```

```
2885 %/mathpiper
```

```
2886     %output,preserve="false"
```

```
2887     Result: True
```

```
2888
```

```
2889     Side Effects:
```

```
2890     50
```

```
2891     51
```

```
2892     52
```

```
2893     53
```

```
2894     54
```

```
2895     55
```

```
2896     56
```

```
2897     57
```

```
2898     58
```

```
2899     59
```

```
2900 .    %/output
```

2901 **15.6 Calculate The Sum Of The Numbers In A List Using ForEach()**

2902 In previous examples, counting code in the form **x := x + 1** was used to count
2903 how many times a while loop was executed. The following program uses a
2904 **ForEach()** function and a line of code similar to this counter to calculate the
2905 **sum of the numbers in a list:**

```
2906 %mathpiper
2907 /*
2908     This program calculates the sum of the numbers
2909     in a list.
2910 */
2911 //This variable is used to accumulate the sum.
2912 sum := 0;
2913 ForEach(x, {1,2,3,4,5,6,7,8,9,10} )
2914 [
2915     /*
2916         Add the contents of x to the contents of sum
2917         and place the result back into sum.
2918     */
2919     sum := sum + x;
2920
2921     //Print the sum as it is being accumulated.
2922     Write(sum,,);
2923 ];
2924 NewLine(); NewLine();
2925 Echo("The sum of the numbers in the list = ", sum);
2926 %/mathpiper
2927 %output,preserve="false"
2928     Result: True
2929
2930     Side Effects:
2931     1,3,6,10,15,21,28,36,45,55,
2932
2933     The sum of the numbers in the list = 55
2934 . %/output
```

2935 In the above program, the integers **1** through **10** were manually placed into a list
2936 by typing them individually. This method is limited because only a relatively
2937 small number of integers can be placed into a list this way. The following section
2938 discusses an operator which can be used to automatically place a large number
2939 of integers into a list with very little typing.

2940 15.7 The .. Range Operator

```
first .. last
```

2941 A programmer often needs to create a list which contains **consecutive integers**
2942 and the **.. "range"** operator can be used to do this. The **first** integer in the list is
2943 placed before the **..** operator and the **last** integer in the list is placed after it
2944 (**Note: there must be a space immediately to the left of the .. operator**
2945 **and a space immediately to the right of it or an error will be generated.**).
2946 Here are some examples:

```
2947 In> 1 .. 10  
2948 Result> {1,2,3,4,5,6,7,8,9,10}
```

```
2949 In> 10 .. 1  
2950 Result> {10,9,8,7,6,5,4,3,2,1}
```

```
2951 In> 1 .. 100  
2952 Result> {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,  
2953         21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,  
2954         38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,  
2955         55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,  
2956         72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,  
2957         89,90,91,92,93,94,95,96,97,98,99,100}
```

```
2958 In> -10 .. 10  
2959 Result> {-10,-9,-8,-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,8,9,10}
```

2960 As these examples show, the **..** operator can generate lists of integers in
2961 ascending order and descending order. It can also generate lists that are very
2962 large and ones that contain negative integers.

2963 Remember, though, if one or both of the spaces around the **..** are omitted, an
2964 error is generated:

```
2965 In> 1..3  
2966 Result>  
2967 Error parsing expression, near token .3.
```

2968 15.8 Using ForEach() With The Range Operator To Print The Prime 2969 Numbers Between 1 And 100

2970 The following program shows how to use a **ForEach()** function instead of a
2971 **While()** function to print the prime numbers between 1 and 100. Notice that
2972 loops that are implemented with **ForEach()** often require less typing than
2973 their **While()** based equivalents:

```
2974 %mathpiper
2975 /*
2976    This program prints the prime integers between 1 and 100 using
2977    a ForEach() function instead of a While() function. Notice that
2978    the ForEach() version requires less typing than the While()
2979    version.
2980 */
2981 ForEach(x, 1 .. 100)
2982 [
2983     If(IsPrime(x), Write(x,,) );
2984 ];
2985 %/mathpiper
2986     %output,preserve="false"
2987     Result: True
2988
2989     Side Effects:
2990     2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,
2991     73,79,83,89,97,
2992 .    %/output
```

2993 15.8.1 Using ForEach() And The Range Operator To Place The Prime 2994 Numbers Between 1 And 50 Into A List

2995 A ForEach() function can also be used to place values in a list, just the the
2996 While() function can:

```
2997 %mathpiper
2998 /*
2999    Place the prime numbers between 1 and 50 into
3000    a list using a ForEach() function.
3001 */
3002 //Create a new list.
3003 primes := {};
3004 ForEach(number, 1 .. 50)
3005 [
3006     /*
3007        If number is prime, append it to the end of the list and
3008        then assign the new list that is created to the variable
3009        'primes'.
3010     */
3011     If(IsPrime(number), primes := Append(primes, number) );
3012 ];
```

```
3013 //Print information about the primes that were found.
3014 Echo("Primes ", primes);
3015 Echo("The number of primes in the list = ", Length(primes) );
3016 Echo("The first number in the list = ", primes[1] );

3017 %/mathpiper

3018     %output,preserve="false"
3019     Result: True
3020
3021     Side Effects:
3022     Primes {2,3,5,7,11,13,17,19,23,29,31,37,41,43,47}
3023     The number of primes in the list = 15
3024     The first number in the list = 2
3025 .    %/output
```

3026 As can be seen from the above examples, the **ForEach()** function and the **range**
3027 **operator** can do a significant amount of work with very little typing. You will
3028 discover in the next section that MathPiper has functions which are even more
3029 powerful than these two.

3030 15.8.2 Exercises

3031 For the following exercises, create a new MathRider worksheet file called
3032 **book_1_section_15b_exercises_<your first name>_<your last name>.mrw**.
3033 (**Note: there are no spaces in this file name**). For example, John Smith's
3034 worksheet would be called:

3035 **book_1_section_15b_exercises_john_smith.mrw**.

3036 After this worksheet has been created, place your answer for each exercise that
3037 requires a fold into its own fold in this worksheet. Place a title attribute in the
3038 start tag of each fold which indicates the exercise the fold contains the solution
3039 to. The folds you create should look similar to this one:

```
3040 %mathpiper,title="Exercise 1"

3041 //Sample fold.

3042 %/mathpiper
```

3043 If an exercise uses the MathPiper console instead of a fold, copy the work you
3044 did in the console into the worksheet so it can be saved.

3045 15.8.3 Exercise 1

3046 Carefully read all of section 15 starting at the end of the previous
3047 exercises and up to this point. Evaluate each one of the examples in the

3048 sections you read in the MathPiper worksheet you created or in the
3049 MathPiper console and verify that the results match the ones in the book.
3050 Copy all of the console examples you evaluated into your worksheet so they
3051 will be saved.

3052 15.8.4 Exercise 2

3053 Create a program that uses a **ForEach()** function and an **IsOdd()** function to
3054 analyze the following list and then print the number of odd numbers it
3055 contains.

3056 {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}

3057 15.8.5 Exercise 3

3058 Create a program that uses a **ForEach()** function and an **IsNegativeNumber()**
3059 function to copy all of the negative numbers in the following list into a
3060 new list. Use the variable **negativeNumbers** to hold the new list.

3061 {36,-29,-33,-6,14,7,-16,-3,-14,37,-38,-8,-45,-21,-26,6,6,38,-20,33,41,-
3062 4,24,37,40,29}

3063 15.8.6 Exercise 4

3064 Create a program that uses a **ForEach()** function to analyze the following
3065 list and then print the following information about it:

- 3066 1) The largest number in the list.
3067 2) The smallest number in the list.
3068 3) The sum of all the numbers in the list.

3069 {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}

3070 15.8.7 Exercise 5

3071 Create a program that uses a **while loop** to make a list that contains **1000**
3072 **random integers** between **1** and **100** inclusive. Then, use a **ForEach()**
3073 function to determine how many integers in the list are **prime** and use an
3074 **Echo()** function to print this total.

3075 **16 Functions & Operators Which Loop Internally**

3076 Looping is such a useful capability that MathPiper has many functions which
3077 loop internally. Now that you have some experience with loops, you can use this
3078 experience to help you imagine how these functions use loops to process the
3079 information that is passed to them.

3080 **16.1 Functions & Operators Which Loop Internally To Process Lists**

3081 This section discusses a number of functions that use loops to process lists.

3082 **16.1.1 TableForm()**

```
TableForm(list)
```

3083 The **TableForm()** function prints the contents of a list in the form of a table.
3084 Each member in the list is printed on its own line and this sometimes makes the
3085 contents of the list easier to read:

```
3086 In> testList := {2,4,6,8,10,12,14,16,18,20}  
3087 Result> {2,4,6,8,10,12,14,16,18,20}
```

```
3088 In> TableForm(testList)  
3089 Result> True  
3090 Side Effects>  
3091 2  
3092 4  
3093 6  
3094 8  
3095 10  
3096 12  
3097 14  
3098 16  
3099 18  
3100 20
```

3101 **16.1.2 Contains()**

3102 The **Contains()** function searches a list to determine if it contains a given
3103 expression. If it finds the expression, it returns **True** and if it doesn't find the
3104 expression, it returns **False**. Here is the calling format for Contains():

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


3105 The following code shows Contains() being used to locate a number in a list:

```
3106 In> Contains({50,51,52,53,54,55,56,57,58,59}, 53)
3107 Result> True
```

```
3108 In> Contains({50,51,52,53,54,55,56,57,58,59}, 75)
3109 Result> False
```

3110 The **Not()** function can also be used with predicate functions like Contains() to
3111 change their results to the opposite truth value:

```
3112 In> Not Contains({50,51,52,53,54,55,56,57,58,59}, 75)
3113 Result> True
```

3114 16.1.3 Find()

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

3115 The **Find()** function searches a list for the first occurrence of a given expression.
3116 If the expression is found, the **position of its first occurrence** is returned and
3117 if it is not found, **-1** is returned:

```
3118 In> Find({23, 15, 67, 98, 64}, 15)
3119 Result> 2
```

```
3120 In> Find({23, 15, 67, 98, 64}, 8)
3121 Result> -1
```

3122 16.1.4 Count()

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

3123 **Count()** determines the number of times a given expression occurs in a list:

```
3124 In> testList := {a,b,b,c,c,c,d,d,d,d,e,e,e,e,e}
3125 Result> {a,b,b,c,c,c,d,d,d,d,e,e,e,e,e}
```

```
3126 In> Count(testList, c)
3127 Result> 3
```

```
3128 In> Count(testList, e)
3129 Result> 5
```

```
3130 In> Count(testList, z)
3131 Result> 0
```

3132 **16.1.5 Select()**

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

3133 **Select()** returns a list that contains all the expressions in a list which make a
3134 given predicate function return **True**:

```
3135 In> Select("IsPositiveInteger", {46,87,59,-27,11,86,-21,-58,-86,-52})  
3136 Result> {46,87,59,11,86}
```

3137 In this example, notice that the **name** of the predicate function is passed to
3138 **Select()** in **double quotes**. There are other ways to pass a predicate function to
3139 **Select()** but these are covered in a later section.

3140 Here are some further examples which use the **Select()** function:

```
3141 In> Select("IsOdd", {16,14,82,92,33,74,99,67,65,52})  
3142 Result> {33,99,67,65}
```

```
3143 In> Select("IsEven", {16,14,82,92,33,74,99,67,65,52})  
3144 Result> {16,14,82,92,74,52}
```

```
3145 In> Select("IsPrime", 1 .. 75)  
3146 Result> {2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73}
```

3147 Notice how the third example uses the **..** operator to automatically generate a list
3148 of consecutive integers from 1 to 75 for the **Select()** function to analyze.

3149 **16.1.6 The Nth() Function & The [] Operator**

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

3150 The **Nth()** function simply returns the expression which is at a given position in
3151 a list. This example shows the **third** expression in a list being obtained:

```
3152 In> testList := {a,b,c,d,e,f,g}  
3153 Result> {a,b,c,d,e,f,g}
```

```
3154 In> Nth(testList, 3)  
3155 Result> c
```

3156 As discussed earlier, the **[]** operator can also be used to obtain a single
3157 expression from a list:

```
3158 In> testList[3]
3159 Result> c
```

3160 The **[]** operator can even obtain a single expression directly from a list without
3161 needing to use a variable:

```
3162 In> {a,b,c,d,e,f,g}[3]
3163 Result> c
```

3164 **16.1.7 The : Prepend Operator**

```
expression : list
```

3165 The prepend operator is a colon **:** and it can be used to add an expression to the
3166 beginning of a list:

```
3167 In> testList := {b,c,d}
3168 Result> {b,c,d}

3169 In> testList := a:testList
3170 Result> {a,b,c,d}
```

3171 **16.1.8 Concat()**

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

3172 The Concat() function is short for "concatenate" which means to join together
3173 sequentially. It takes two or more lists and joins them together into a single
3174 larger list:

```
3175 In> Concat({a,b,c}, {1,2,3}, {x,y,z})
3176 Result> {a,b,c,1,2,3,x,y,z}
```

3177 **16.1.9 Insert(), Delete(), & Replace()**

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

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

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

3178 **Insert()** inserts an expression into a list at a given index, **Delete()** deletes an
3179 expression from a list at a given index, and **Replace()** replaces an expression in
3180 a list at a given index with another expression:

```
3181 In> testList := {a,b,c,d,e,f,g}
3182 Result> {a,b,c,d,e,f,g}

3183 In> testList := Insert(testList, 4, 123)
3184 Result> {a,b,c,123,d,e,f,g}

3185 In> testList := Delete(testList, 4)
3186 Result> {a,b,c,d,e,f,g}

3187 In> testList := Replace(testList, 4, xxx)
3188 Result> {a,b,c,xxx,e,f,g}
```

3189 16.1.10 Take()

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

3190 **Take()** obtains a sublist from the **beginning** of a list, the **end** of a list, or the
3191 **middle** of a list. The expressions in the list that are not taken are discarded.

3192 A **positive** integer passed to Take() indicates how many expressions should be
3193 taken from the **beginning** of a list:

```
3194 In> testList := {a,b,c,d,e,f,g}
3195 Result> {a,b,c,d,e,f,g}

3196 In> Take(testList, 3)
3197 Result> {a,b,c}
```

3198 A **negative** integer passed to Take() indicates how many expressions should be
3199 taken from the **end** of a list:

```
3200 In> Take(testList, -3)
3201 Result> {e,f,g}
```

3202 Finally, if a **two member list** is passed to Take() it indicates the **range** of
3203 expressions that should be taken from the **middle** of a list. The **first** value in the
3204 passed-in list specifies the **beginning** index of the range and the **second** value
3205 specifies its **end**:

```
3206 In> Take(testList, {3,5})
3207 Result> {c,d,e}
```

3208 **16.1.11 Drop()**

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

3209 **Drop()** does the opposite of Take() in that it **drops** expressions from the
3210 **beginning** of a list, the **end** of a list, or the **middle** of a list and **returns a list**
3211 **which contains the remaining expressions.**

3212 A **positive** integer passed to Drop() indicates how many expressions should be
3213 dropped from the **beginning** of a list:

```
3214 In> testList := {a,b,c,d,e,f,g}
3215 Result> {a,b,c,d,e,f,g}
```

```
3216 In> Drop(testList, 3)
3217 Result> {d,e,f,g}
```

3218 A **negative** integer passed to Drop() indicates how many expressions should be
3219 dropped from the **end** of a list:

```
3220 In> Drop(testList, -3)
3221 Result> {a,b,c,d}
```

3222 Finally, if a **two member list** is passed to Drop() it indicates the **range** of
3223 expressions that should be dropped from the **middle** of a list. The **first** value in
3224 the passed-in list specifies the **beginning** index of the range and the **second**
3225 value specifies its **end**:

```
3226 In> Drop(testList, {3,5})
3227 Result> {a,b,f,g}
```

3228 **16.1.12 FillList()**

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

3229 The FillList() function simply creates a list which is of size "length" and fills it
3230 with "length" copies of the given expression:

```
3231 In> FillList(a, 5)
3232 Result> {a,a,a,a,a}
```

```
3233 In> FillList(42,8)
3234 Result> {42,42,42,42,42,42,42,42}
```

3235 **16.1.13 RemoveDuplicates()**

```
RemoveDuplicates(list)
```

3236 **RemoveDuplicates()** removes any duplicate expressions that are contained in a
3237 list:

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

```
3239 Result> {a,a,b,c,c,b,b,a,b,c,c}
```

```
3240 In> RemoveDuplicates(testList)
```

```
3241 Result> {a,b,c}
```

3242 **16.1.14 Reverse()**

```
Reverse(list)
```

3243 **Reverse()** reverses the order of the expressions in a list:

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

```
3245 Result> {a,b,c,d,e,f,g,h}
```

```
3246 In> Reverse(testList)
```

```
3247 Result> {h,g,f,e,d,c,b,a}
```

3248 **16.1.15 Partition()**

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

3249 The **Partition()** function breaks a list into sublists of size "partition_size":

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

```
3251 Result> {a,b,c,d,e,f,g,h}
```

```
3252 In> Partition(testList, 2)
```

```
3253 Result> {{a,b},{c,d},{e,f},{g,h}}
```

3254 If the partition_size does not divide the length of the list **evenly**, the remaining
3255 elements are discarded:

```
3256 In> Partition(testList, 3)
```

```
3257 Result> {{h,b,c},{d,e,f}}
```

3258 The number of elements that Partition() will discard can be calculated by
3259 dividing the length of a list by the partition size and obtaining the **remainder**:

```
3260 In> Length(testList) % 3  
3261 Result> 2
```

3262 Remember that % is the remainder operator. It divides two integers and returns
3263 their remainder.

3264 16.1.16 Table()

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

3265 The Table() function creates a list of values by doing the following:

- 3266 1) Generating a sequence of values between a "begin_value" and an
3267 "end_value" with each value being incremented by the "step_amount".
- 3268 2) Placing each value in the sequence into the specified "variable", one value
3269 at a time.
- 3270 3) Evaluating the defined "expression" (which contains the defined "variable")
3271 for each value, one at a time.
- 3272 4) Placing the result of each "expression" evaluation into the result list.

3273 This example generates a list which contains the integers 1 through 10:

```
3274 In> Table(x, x, 1, 10, 1)  
3275 Result> {1,2,3,4,5,6,7,8,9,10}
```

3276 Notice that the expression in this example is simply the variable 'x' itself with no
3277 other operations performed on it.

3278 The following example is similar to the previous one except that its expression
3279 multiplies 'x' by 2:

```
3280 In> Table(x*2, x, 1, 10, 1)  
3281 Result> {2,4,6,8,10,12,14,16,18,20}
```

3282 Lists which contain decimal values can also be created by setting the
3283 "step_amount" to a decimal:

```
3284 In> Table(x, x, 0, 1, .1)  
3285 Result> {0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1}
```

3286 **16.1.17 HeapSort()**

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

3287 **HeapSort()** sorts the elements of **list** into the order indicated by **compare** with
3288 **compare** typically being the **less than** operator "<" or the **greater than**
3289 operator ">":

```
3290 In> HeapSort({4,7,23,53,-2,1}, "<");  
3291 Result: {-2,1,4,7,23,53}
```

```
3292 In> HeapSort({4,7,23,53,-2,1}, ">");  
3293 Result: {53,23,7,4,1,-2}
```

```
3294 In> HeapSort({1/2,3/5,7/8,5/16,3/32}, "<")  
3295 Result: {3/32,5/16,1/2,3/5,7/8}
```

```
3296 In> HeapSort({.5,3/5,.76,5/16,3/32}, "<")  
3297 Result: {3/32,5/16,.5,3/5,.76}
```

3298 **16.2 Functions That Work With Integers**

3299 This section discusses various functions which work with integers. Some of
3300 these functions also work with non-integer values and their use with non-
3301 integers is discussed in other sections.

3302 **16.2.1 RandomIntegerVector()**

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

3303 A vector is a list that does not contain other lists. **RandomIntegerVector()**
3304 creates a list of size "length" that contains random integers that are no lower
3305 than "lowest_possible" and no higher than "highest possible". The following
3306 example creates **10** random integers between **1** and **99** inclusive:

```
3307 In> RandomIntegerVector(10, 1, 99)  
3308 Result> {73,93,80,37,55,93,40,21,7,24}
```

3309 **16.2.2 Max() & Min()**

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

3310 If two values are passed to **Max()**, it determines which one is larger:

```
3311 In> Max(10, 20)
```


3312 `Result> 20`

3313 If a list of values are passed to `Max()`, it finds the largest value in the list:

3314 `In> testList := RandomIntegerVector(10, 1, 99)`

3315 `Result> {73,93,80,37,55,93,40,21,7,24}`

3316 `In> Max(testList)`

3317 `Result> 93`

3318 The **Min()** function is the opposite of the `Max()` function.

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

3319 If two values are passed to `Min()`, it determines which one is smaller:

3320 `In> Min(10, 20)`

3321 `Result> 10`

3322 If a list of values are passed to `Min()`, it finds the smallest value in the list:

3323 `In> testList := RandomIntegerVector(10, 1, 99)`

3324 `Result> {73,93,80,37,55,93,40,21,7,24}`

3325 `In> Min(testList)`

3326 `Result> 7`

3327 **16.2.3 Div() & Mod()**

```
Div(dividend, divisor)
Mod(dividend, divisor)
```

3328 **Div()** stands for "divide" and determines the whole number of times a divisor
3329 goes into a dividend:

3330 `In> Div(7, 3)`

3331 `Result> 2`

3332 **Mod()** stands for "modulo" and it determines the remainder that results when a
3333 dividend is divided by a divisor:

3334 `In> Mod(7,3)`

3335 `Result> 1`

3336 The remainder/modulo operator **%** can also be used to calculate a remainder:

```
3337 In> 7 % 2
3338 Result> 1
```

3339 **16.2.4 Gcd()**

```
Gcd(value1, value2)
Gcd(list)
```

3340 GCD stands for Greatest Common Divisor and the **Gcd()** function determines the
3341 greatest common divisor of the values that are passed to it.

3342 If two integers are passed to Gcd(), it calculates their greatest common divisor:

```
3343 In> Gcd(21, 56)
3344 Result> 7
```

3345 If a list of integers are passed to Gcd(), it finds the greatest common divisor of all
3346 the integers in the list:

```
3347 In> Gcd({9, 66, 123})
3348 Result> 3
```

3349 **16.2.5 Lcm()**

```
Lcm(value1, value2)
Lcm(list)
```

3350 LCM stands for Least Common Multiple and the **Lcm()** function determines the
3351 least common multiple of the values that are passed to it.

3352 If two integers are passed to Lcm(), it calculates their least common multiple:

```
3353 In> Lcm(14, 8)
3354 Result> 56
```

3355 If a list of integers are passed to Lcm(), it finds the least common multiple of all
3356 the integers in the list:

```
3357 In> Lcm({3, 7, 9, 11})
3358 Result> 693
```

3359 **16.2.6 Sum()**

```
Sum(list)
```

3360 **Sum()** can find the sum of a list that is passed to it:

3361 In> testList := RandomIntegerVector(10,1,99)

3362 Result> {73,93,80,37,55,93,40,21,7,24}

3363 In> Sum(testList)

3364 Result> 523

3365 In> testList := 1 .. 10

3366 Result> {1,2,3,4,5,6,7,8,9,10}

3367 In> Sum(testList)

3368 Result> 55

3369 **16.2.7 Product()**

```
Product(list)
```

3370 This function has two calling formats, only one of which is discussed here.

3371 **Product(list)** multiplies all the expressions in a list together and returns their
3372 product:

3373 In> Product({1,2,3})

3374 Result> 6

3375 **16.3 Exercises**

3376 For the following exercises, create a new MathRider worksheet file called
3377 **book_1_section_16_exercises_<your first name>_<your last name>.mrw.**
3378 (**Note: there are no spaces in this file name**). For example, John Smith's
3379 worksheet would be called:

3380 **book_1_section_16_exercises_john_smith.mrw.**

3381 After this worksheet has been created, place your answer for each exercise that
3382 requires a fold into its own fold in this worksheet. Place a title attribute in the
3383 start tag of each fold which indicates the exercise the fold contains the solution
3384 to. The folds you create should look similar to this one:

3385 %mathpiper,title="Exercise 1"

3386 //Sample fold.

3387 [%/mathpiper](#)

3388 If an exercise uses the MathPiper console instead of a fold, copy the work you
3389 did in the console into the worksheet so it can be saved.

3390 **16.3.1 Exercise 1**

3391 Carefully read all of section 16 up to this point. Evaluate each one of
3392 the examples in the sections you read in the MathPiper worksheet you
3393 created or in the MathPiper console and verify that the results match the
3394 ones in the book. Copy all of the console examples you evaluated into your
3395 worksheet so they will be saved.

3396 **16.3.2 Exercise 2**

3397 Create a program that uses **RandomIntegerVector()** to create a 100 member
3398 list that contains random integers between 1 and 5 inclusive. Use **Count()**
3399 to determine how many of each digit 1-5 are in the list and then print this
3400 information. Hint: you can use the **HeapSort()** function to sort the
3401 generated list to make it easier to check if your program is counting
3402 correctly.

3403 **16.3.3 Exercise 3**

3404 Create a program that uses **RandomIntegerVector()** to create a 100 member
3405 list that contains random integers between 1 and 50 inclusive and use
3406 **Contains()** to determine if the number 25 is in the list. Print "25 was in
3407 the list." if 25 was found in the list and "25 was not in the list." if it
3408 wasn't found.

3409 **16.3.4 Exercise 4**

3410 Create a program that uses **RandomIntegerVector()** to create a 100 member
3411 list that contains random integers between 1 and 50 inclusive and use
3412 **Find()** to determine if the number 10 is in the list. Print the position of
3413 10 if it was found in the list and "10 was not in the list." if it wasn't
3414 found.

3415 **16.3.5 Exercise 5**

3416 Create a program that uses **RandomIntegerVector()** to create a 100 member
3417 list that contains random integers between 0 and 3 inclusive. Use **Select()**
3418 with the **IsNonZeroInteger()** predicate function to obtain all of the nonzero
3419 integers in this list.

3420 **16.3.6 Exercise 6**

3421 Create a program that uses **Table()** to obtain a list which contains the
3422 squares of the integers between 1 and 10 inclusive.

3423 17 Nested Loops

3424 Now that you have seen how to solve problems with single loops, it is time to
3425 discuss what can be done when a loop is placed inside of another loop. A loop
3426 that is placed **inside** of another loop it is called a **nested loop** and this nesting
3427 can be extended to numerous levels if needed. This means that loop 1 can have
3428 loop 2 placed inside of it, loop 2 can have loop 3 placed inside of it, loop 3 can
3429 have loop 4 placed inside of it, and so on.

3430 Nesting loops allows the programmer to accomplish an enormous amount of
3431 work with very little typing.

3432 17.1 Generate All The Combinations That Can Be Entered Into A Two Digit 3433 Wheel Lock Using A Nested Loop



3434 The following program generates all the combinations that can be entered into a
3435 two digit wheel lock. It uses a nested loop to accomplish this with the "**inside**"
3436 nested loop being used to generate **one's place** digits and the "**outside**" loop
3437 being used to generate **ten's place** digits.

```
3438 %mathpiper
3439 /*
3440  Generate all the combinations can be entered into a two
3441  digit wheel lock.
3442  */
3443 combinations := {};
3444 ForEach(digit1, 0 .. 9) //This loop is called the "outside" loop.
```

```
3445 [
3446     ForEach(digit2, 0 .. 9)//This loop is called the "inside" loop.
3447     [
3448         combinations := Append(combinations, {digit1, digit2});
3449     ];
3450 ];

3451 Echo(TableForm(combinations));

3452 %/mathpiper

3453     %output,preserve="false"
3454     Result: True
3455
3456     Side Effects:
3457     {0,0}
3458     {0,1}
3459     {0,2}
3460     {0,3}
3461     {0,4}
3462     {0,5}
3463     {0,6}
3464     .
3465     . //The middle of the list has not been shown.
3466     .
3467     {9,3}
3468     {9,4}
3469     {9,5}
3470     {9,6}
3471     {9,7}
3472     {9,8}
3473     {9,9}
3474     True
3475 . %/output
```

3476 The relationship between the outside loop and the inside loop is interesting
3477 because each time the **outside loop cycles once**, the **inside loop cycles 10**
3478 **times**. Study this program carefully because nested loops can be used to solve a
3479 wide range of problems and therefore understanding how they work is
3480 important.

3481 17.2 Exercises

3482 For the following exercises, create a new MathRider worksheet file called
3483 **book_1_section_17_exercises_<your first name>_<your last name>.mrw**.
3484 (**Note: there are no spaces in this file name**). For example, John Smith's
3485 worksheet would be called:

3486 **book_1_section_17_exercises_john_smith.mrw**.

3487 After this worksheet has been created, place your answer for each exercise that
3488 requires a fold into its own fold in this worksheet. Place a title attribute in the
3489 start tag of each fold which indicates the exercise the fold contains the solution
3490 to. The folds you create should look similar to this one:

3491 `%mathpiper,title="Exercise 1"`

3492 `//Sample fold.`

3493 `%/mathpiper`

3494 If an exercise uses the MathPiper console instead of a fold, copy the work you
3495 did in the console into the worksheet so it can be saved.

3496 **17.2.1 Exercise 1**

3497 Carefully read all of section 17 up to this point. Evaluate each one of
3498 the examples in the sections you read in the MathPiper worksheet you
3499 created or in the MathPiper console and verify that the results match the
3500 ones in the book. Copy all of the console examples you evaluated into your
3501 worksheet so they will be saved.

3502 **17.2.2 Exercise 2**

3503 Create a program that will generate all of the combinations that can be
3504 entered into a three digit wheel lock. (Hint: a triple nested loop can be
3505 used to accomplish this.)

3506 18 User Defined Functions

3507 In computer programming, a **function** is a named section of code that can be
3508 **called** from other sections of code. **Values** can be sent to a function for
3509 processing as part of the **call** and a function always returns a value as its result.
3510 A function can also generate side effects when it is called and side effects have
3511 been covered in earlier sections.

3512 The values that are sent to a function when it is called are called **arguments** or
3513 **actual parameters** and a function can accept 0 or more of them. These
3514 arguments are placed within parentheses.

3515 MathPiper has many predefined functions (some of which have been discussed in
3516 previous sections) but users can create their own functions too. The following
3517 program creates a function called **addNums()** which takes two numbers as
3518 arguments, adds them together, and returns their sum back to the calling code
3519 as a result:

```
3520 In> addNums(num1,num2) := num1 + num2  
3521 Result> True
```

3522 This line of code defined a new function called **addNums** and specified that it
3523 will accept two values when it is called. The **first** value will be placed into the
3524 variable **num1** and the **second** value will be placed into the variable **num2**.

3525 Variables like num1 and num2 which are used in a function to accept values from
3526 calling code are called **formal parameters**. **Formal parameter variables** are
3527 used inside a function to process the **values/actual parameters/arguments**
3528 that were placed into them by the calling code.

3529 The code on the **right side** of the **assignment operator** is **bound** to the
3530 function name "**addNums**" and it is executed each time **addNums()** is called.
3531 The following example shows the new **addNums()** function being called multiple
3532 times with different values being passed to it:

```
3533 In> addNums(2,3)  
3534 Result> 5
```

```
3535 In> addNums(4,5)  
3536 Result> 9
```

```
3537 In> addNums(9,1)  
3538 Result> 10
```

3539 Notice that, unlike the functions that come with MathPiper, we chose to have this
3540 function's name start with a **lower case letter**. We could have had addNums()
3541 begin with an upper case letter but it is a **convention** in MathPiper for **user**

3542 **defined function names to begin with a lower case letter to distinguish**
3543 **them from the functions that come with MathPiper.**

3544 The values that are returned from user defined functions can also be assigned to
3545 variables. The following example uses a %mathpiper fold to define a function
3546 called **evenIntegers()** and then this function is used in the MathPiper console:

```
3547 %mathpiper
3548 evenIntegers(endInteger) :=
3549 [
3550     resultList := {};
3551     x := 2;
3552     While(x <= endInteger)
3553     [
3554         resultList := Append(resultList, x);
3555         x := x + 2;
3556     ];
3557     /*
3558     The result of the last expression which is executed in a function
3559     is the result that the function returns to the caller. In this case,
3560     resultList is purposely being executed last so that its contents are
3561     returned to the caller.
3562     */
3563     resultList;
3564 ];
3565
3566 %/mathpiper
3567
3568 %output,preserve="false"
3569 Result: True
3570 . %/output
3571
3571 In> a := evenIntegers(10)
3572 Result> {2,4,6,8,10}
3573
3573 In> Length(a)
3574 Result> 5
```

3575 The function **evenIntegers()** returns a list which contains all the even integers
3576 from 2 up through the value that was passed into it. The fold was first executed
3577 in order to define the **evenIntegers()** function and make it ready for use. The
3578 **evenIntegers()** function was then called from the MathPiper console and 10
3579 was passed to it.

3580 After the function was finished executing, it returned a list of even integers as a

3581 result and this result was assigned to the variable 'a'. We then passed the list
3582 that was assigned to 'a' to the **Length()** function in order to determine its size.

3583 **18.1 Global Variables, Local Variables, & Local()**

3584 The new **evenIntegers()** function seems to work well, but there is a problem.
3585 The variables '**x**' and **resultList** were defined inside the function as **global**
3586 **variables** which means they are accessible from anywhere, including from
3587 within other functions, within other folds (as shown here):

```
3588 %mathpiper
3589 Echo(x, ",", resultList);
3590 %/mathpiper
3591     %output,preserve="false"
3592     Result: True
3593
3594     Side Effects:
3595     12 , {2,4,6,8,10}
3596 .    %/output
```

3597 and from within the MathPiper console:

```
3598 In> x
3599 Result> 12
3600 In> resultList
3601 Result> {2,4,6,8,10}
```

3602 **Using global variables inside of functions is usually not a good idea**
3603 because code in other functions and folds might already be using (or will use) the
3604 same variable names. Global variables which have the same name are the same
3605 variable. When one section of code changes the value of a given global variable,
3606 the value is changed everywhere that variable is used and this will eventually
3607 cause problems.

3608 In order to prevent errors being caused by global variables having the same
3609 name, a function named **Local()** can be called inside of a function to define what
3610 are called **local variables**. A **local variable** is only accessible inside the
3611 function it has been defined in, even if it has the same name as a global variable.
3612 The following example shows a second version of the **evenIntegers()** function
3613 which uses **Local()** to make '**x**' and **resultList** local variables:

```
3614 %mathpiper
3615 /*
3616  This version of evenIntegers() uses Local() to make
3617  x and resultList local variables
3618 */
3619 evenIntegers(endInteger) :=
3620 [
3621     Local(x,resultList);
3622     resultList := {};
3623
3624     x := 2;
3625
3626     While(x <= endInteger)
3627     [
3628         resultList := Append(resultList, x);
3629         x := x + 2;
3630     ];
3631
3632     /*
3633     The result of the last expression which is executed in a function
3634     is the result that the function returns to the caller. In this case,
3635     resultList is purposely being executed last so that its contents are
3636     returned to the caller.
3637     */
3638     resultList;
3639 ];
3640 %/mathpiper
3641     %output,preserve="false"
3642     Result: True
3643 . %/output
```

3644 We can verify that '**x**' and **resultList** are now local variables by first clearing
3645 them, calling **evenIntegers()**, and then seeing what '**x**' and **resultList** contain:

```
3646 In> Clear(x, resultList)
3647 Result> True
3648 In> evenIntegers(10)
3649 Result> {2,4,6,8,10}
3650 In> x
3651 Result> x
3652 In> resultList
3653 Result> resultList
```

3654 18.2 Exercises

3655 For the following exercises, create a new MathRider worksheet file called
3656 **book_1_section_18_exercises_<your first name>_<your last name>.mrw.**
3657 **(Note: there are no spaces in this file name).** For example, John Smith's
3658 worksheet would be called:

3659 **book_1_section_18_exercises_john_smith.mrw.**

3660 After this worksheet has been created, place your answer for each exercise that
3661 requires a fold into its own fold in this worksheet. Place a title attribute in the
3662 start tag of each fold which indicates the exercise the fold contains the solution
3663 to. The folds you create should look similar to this one:

```
3664 %mathpiper,title="Exercise 1"
```

```
3665 //Sample fold.
```

```
3666 %/mathpiper
```

3667 If an exercise uses the MathPiper console instead of a fold, copy the work you
3668 did in the console into the worksheet so it can be saved.

3669 18.2.1 Exercise 1

3670 Carefully read all of section 18 up to this point. Evaluate each one of
3671 the examples in the sections you read in the MathPiper worksheet you
3672 created or in the MathPiper console and verify that the results match the
3673 ones in the book. Copy all of the console examples you evaluated into your
3674 worksheet so they will be saved.

3675 18.2.2 Exercise 2

3676 Create a function called **tenOddIntegers()** which returns a list which
3677 contains 10 random odd integers between 1 and 99 inclusive.

3678 18.2.3 Exercise 3

3679 Create a function called **convertStringToList(string)** which takes a string
3680 as a parameter and returns a list which contains all of the characters in
3681 the string. Here is an example of how the function should work:

```
3682 In> convertStringToList("Hello friend!")  
3683 Result> {"H","e","l","l","o"," ","f","r","i","e","n","d","!"}
```

```
3684 In> convertStringToList("Computer Algebra System")  
3685 Result> {"C","o","m","p","u","t","e","r"," ","A","l","g","e","b","r","a","  
3686 ","S","y","s","t","e","m"}
```

3687 19 Miscellaneous topics

3688 19.1 Incrementing And Decrementing Variables With The ++ And -- 3689 Operators

3690 Up until this point we have been adding 1 to a variable with code in the form of **x**
3691 **:= x + 1** and subtracting 1 from a variable with code in the form of **x := x - 1**.
3692 Another name for **adding** 1 to a variable is **incrementing** it and **decrementing**
3693 a variable means to **subtract** 1 from it. Now that you have had some experience
3694 with these longer forms, it is time to show you shorter versions of them.

3695 19.1.1 Incrementing Variables With The ++ Operator

3696 The number 1 can be added to a variable by simply placing the ++ operator after
3697 it like this:

```
3698 In> x := 1  
3699 Result: 1
```

```
3700 In> x++;  
3701 Result: True
```

```
3702 In> x  
3703 Result: 2
```

3704 Here is a program that uses the ++ operator to increment a loop index variable:

```
3705 %mathpiper  
3706 count := 1;  
3707 While(count <= 10)  
3708 [  
3709     Echo(count);  
3710  
3711     count++; //The ++ operator increments the count variable.  
3712 ];  
3713 %/mathpiper  
3714 %output,preserve="false"  
3715 Result: True  
3716  
3717 Side Effects:  
3718 1  
3719 2
```

```
3720      3
3721      4
3722      5
3723      6
3724      7
3725      8
3726      9
3727     10
3728 .    %/output
```

3729 19.1.2 Decrementing Variables With The -- Operator

3730 The number 1 can be subtracted from a variable by simply placing the --
3731 operator after it like this:

```
3732 In> x := 1
3733 Result: 1

3734 In> x--;
3735 Result: True

3736 In> x
3737 Result: 0
```

3738 Here is a program that uses the -- operator to decrement a loop index variable:

```
3739 %mathpiper

3740 count := 10;

3741 While(count >= 1)
3742 [
3743     Echo(count);
3744     count--; //The -- operator decrements the count variable.
3745 ];

3747 %/mathpiper

3748 %output,preserve="false"
3749 Result: True
3750
3751 Side Effects:
3752 10
3753 9
3754 8
3755 7
3756 6
3757 5
```

```
3758      4
3759      3
3760      2
3761      1
3762 .    %/output
```

3763 19.2 Exercises

3764 For the following exercises, create a new MathRider worksheet file called
3765 **book_1_section_19_exercises_<your first name>_<your last name>.mrw.**
3766 **(Note: there are no spaces in this file name).** For example, John Smith's
3767 worksheet would be called:

3768 **book_1_section_19_exercises_john_smith.mrw.**

3769 After this worksheet has been created, place your answer for each exercise that
3770 requires a fold into its own fold in this worksheet. Place a title attribute in the
3771 start tag of each fold which indicates the exercise the fold contains the solution
3772 to. The folds you create should look similar to this one:

```
3773 %mathpiper,title="Exercise 1"
```

```
3774 //Sample fold.
```

```
3775 %/mathpiper
```

3776 If an exercise uses the MathPiper console instead of a fold, copy the work you
3777 did in the console into the worksheet so it can be saved.

3778 19.2.1 Exercise 1

3779 Carefully read all of section 19 up to this point. Evaluate each one of
3780 the examples in the sections you read in the MathPiper worksheet you
3781 created or in the MathPiper console and verify that the results match the
3782 ones in the book. Copy all of the console examples you evaluated into your
3783 worksheet so they will be saved.