

Introduction To Programming With MathRider And MathPiper

by Ted Kosan

Copyright © 2009 by Ted Kosan

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

Table of Contents

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

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

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

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

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

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

1 Preface

2 1.1 Dedication

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

5 1.2 Acknowledgments

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

8 Susan Addington

9 Matthew Moelter

10 Sherm Ostrowsky

11 1.3 Support Email List

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

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

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

22 **2 Introduction**

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

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

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

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

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

49 http://en.wikipedia.org/wiki/Comparison_of_computer_algebra_systems

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

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:

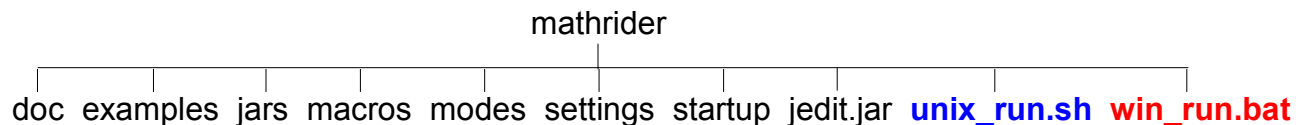


Illustration 1: MathRider's Directory Structure

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
427 them. The purpose of the **N()** function is to make sure that the information that
428 is sent to it is processed numerically instead of symbolically. Functions are used
429 by **evaluating** them and this happens when <shift><enter> is pressed. Another
430 name for evaluating a function is **calling** it.

431 5.2.1.1 The Sqrt() Square Root Function

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


```
434 In> Sqrt(9)
435 Result: 3

436 In> Sqrt(8)
437 Result: Sqrt(8)

438 In> N(Sqrt(8))
439 Result: 2.828427125
```

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

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

```
448     2.828427125
449     2.82842712474619
450     2.82842712474619009760337744842
451     2.8284271247461900976033774484193961571393437507539
```

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

456 **5.2.1.2 The IsEven() Function**

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

```
459 In> IsEven(4)
460 Result> True

461 In> IsEven(5)
462 Result> False
```

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

468 **5.2.2 Accessing Previous Input And Results**

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

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

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

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

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

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

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

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

491 **5.3.1 Syntax Errors**

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

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

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

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

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

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

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

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

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

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

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

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

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

526 **5.4.1 Variables**

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

533 'totalAmount', and 'loop6'.

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

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

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

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

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

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

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

550 **was bound to (in this case $\frac{5}{6}$) was returned.** This value will stay bound to
551 the variable 'a' as long as MathPiper is running unless 'a' is cleared with the
552 **Clear()** function or 'a' has another value assigned to it. This is why we were able
553 to determine both the numerator and the denominator of the rational number
554 assigned to 'a' using two functions in turn.

555 **5.4.1.1 Calculating With Unbound Variables**

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

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

```
559 In> a
560 Result> 9
```

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

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

```
565 In> a
566 Result> a
```

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

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

```
576 In> Factor(8)
577 Result> 2^3
```

```
578 In> Factor(14)
579 Result> 2*7
```

```
580 In> Factor(2343)
581 Result> 3*11*71
```

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

```
583 In> x
584 Result> x
```

```
585 In> IsBound(x)
586 Result> False
```

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

```
589 In> Expand(%)
590 Result> x^2+24*x+80
```

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

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

601 parentheses to an expression and the '**x**' in **Expand** can be thought of **xing** out
602 or removing parentheses from an expression.

603 **5.4.1.2 Variable And Function Names Are Case Sensitive**

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

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

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

```
614 In> Box
615 Result> 1
```

```
616 In> box
617 Result> 2
```

618 **5.4.1.3 Using More Than One Variable**

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

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

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

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

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

```
629 answer
630 Result> 5
```

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

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

639 **5.5 Exercises**

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

642 **5.5.1 Exercise 1**

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

646 **5.5.2 Exercise 2**

647 Answer each one of the following questions:

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

649 b) What is a variable?

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

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

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

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

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

655 h) What does the % character do?

656 **5.5.3 Exercise 3**

657 Perform the following calculation:

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

658 **5.5.4 Exercise 4**

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

660 using the following line of code:

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

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

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

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

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

668 In> **answer**

669 **5.5.5 Exercise 5**

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

672 a)

673 In> x

674 b)

675 In> y

676 c)

677 In> z

678 d)

679 In> x + y

680 e)

681 In> x + z

682 f)

683 In> x + y + z

684 **6 The MathPiper Documentation Plugin**

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

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

693 **6.1 Function List**

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

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

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

717 **6.2 Mini Web Browser Interface**

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

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

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

728 **6.3 Exercises**

729 **6.3.1 Exercise 1**

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

735 **6.3.2 Exercise 2**

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

741 **7 Using MathRider As A Programmer's Text Editor**

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

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

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

755 **7.1 Creating, Opening, Saving, And Closing Text Files**

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

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

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

768 **7.2 Editing Files**

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

774 **7.3 File Modes**

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

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

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

784 ***7.4 Learning How To Type Properly Is An Excellent Investment Of Your*** 785 ***Time***

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

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

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

796 ***7.5 Exercises***

797 ***7.5.1 Exercise 1***

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

802 **8 MathRider Worksheet Files**

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

810 **8.1 Code Folds**

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

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

```
821 %mathpiper
822 2 + 3;
823 %/mathpiper
```

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

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

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

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

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

841 **8.1.1 The title Attribute**

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

```
845 %mathpiper,title="Add two numbers together."
```

```
846 2 + 3;
```

```
847 %/mathpiper
```

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

852 **8.2 Automatically Inserting Folds & Removing Unpreserved Folds**

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

```
854 %mathpiper
```

```
855 %/mathpiper
```

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

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

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

866 **8.3 Placing Text Outside Of A Fold**

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

868 Text can be placed above folds like this.

```
869 text text text text
```

```
870 text text text text
```

```
871 %mathpiper,title="Fold 1"
```

```
872 2 + 3;
```

```
873 %/mathpiper
```

874 Text can be placed between folds like this.

```
875 text text text text
```

```
876 text text text text
```

```
877 %mathpiper,title="Fold 2"
```

```
878 3 + 4;
```

```
879 %/mathpiper
```

880 Text can be placed between folds like this.

```
881 text text text text
```

```
882 text text text text
```

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

886 **8.4 Exercises**

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

889 [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)
890 [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)

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

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

898 **8.4.1 Exercise 1**

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

902 **8.4.2 Exercise 2**

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

905 **8.4.3 Exercise 3**

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

907 **8.4.4 Exercise 4**

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

911 a) $2 * 7 + 3$

912 b) $18 / 3$

913 c) $234238342 + 2038408203$

914 d) $324802984 * 2308098234$

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

916 **9 MathPiper Programming Fundamentals**

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

921 **9.1 Values and Expressions**

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

924 `3`

925 represents the number three, the value:

926 `0.5`

927 represents the number one half, and the value:

928 `"Mathematics is powerful!"`

929 represents an English sentence.

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

933 `3`

934 `2 + 3`

935 `5 + 6*21/18 - 2^3`

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

939 `In> 2 + 3`

940 `Result> 5`

941 **9.2 Operators**

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

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

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

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

952 In> 5 - 2
953 Result> 3

954 In> 3*4
955 Result> 12

956 In> 30/3
957 Result> 10

958 In> 8%5
959 Result> 3

960 In> 2^3
961 Result> 8

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

963 In> -3
964 Result> -3

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

967 In> - -3
968 Result> 3

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

974 **9.3 Operator Precedence**

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

982 [^] Exponents are evaluated right to left.
 983 *,%,/ Then multiplication, remainder, and division operations are evaluated
 984 left to right.
 985 +, - Finally, addition and subtraction are evaluated left to right.

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

988 5 + 6*21/18 - 2^3

989 And here it is in traditional form:

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

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

992 5 + 6*21/18 - 2^3
 993 5 + 6*21/18 - 8
 994 5 + 126/18 - 8
 995 5 + 7 - 8
 996 12 - 8
 997 4

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

1002 **9.4 Changing The Order Of Operations In An Expression**

1003 The default order of operations for an expression can be changed by grouping
 1004 various parts of the expression within parentheses (). Parentheses force the
 1005 code that is placed inside of them to be evaluated before any other operators are
 1006 evaluated. For example, the expression 2 + 4*5 evaluates to 22 using the
 1007 default precedence rules:

1008 In> 2 + 4*5
 1009 Result> 22

1010 If parentheses are placed around 2 + 4, however, the addition operator is forced
 1011 to be evaluated before the multiplication operator and the result is 30:

```
1012 In> (2 + 4)*5
1013 Result> 30
```

1014 Parentheses can also be nested and nested parentheses are evaluated from the
1015 most deeply nested parentheses outward:

```
1016 In> ((2 + 4)*3)*5
1017 Result> 90
```

1018 (Note: precedence adjusting parentheses are different from the parentheses that
1019 are used to call functions.)

1020 Since parentheses are evaluated before any other operators, they are placed at
1021 the top of the precedence table:

- 1022 () Parentheses are evaluated from the inside out.
- 1023 ^ Then exponents are evaluated right to left.
- 1024 *,%,/ Then multiplication, remainder, and division operations are evaluated
1025 left to right.
- 1026 +, − Finally, addition and subtraction are evaluated left to right.

1027 **9.5 Functions & Function Names**

1028 In programming, **functions** are named blocks of code that can be executed one
1029 or more times by being **called** from other parts of the same program or called
1030 from other programs. Functions **can have values passed to them** from the
1031 calling code and they **always return a value** back to the calling code when they
1032 are finished executing. An example of a function is the **IsEven()** function which
1033 was discussed in an previous section.

1034 Functions are one way that MathPiper enables code to be reused. Most
1035 programming languages allow code to be reused in this way, although in other
1036 languages these named blocks of code are sometimes called **subroutines**,
1037 **procedures**, or **methods**.

1038 The functions that come with MathPiper have names which consist of either a
1039 single word (such as **Sum()**) or multiple words that have been put together to
1040 form a compound word (such as **IsBound()**). All letters in the names of
1041 functions which come with MathPiper are lower case except the beginning letter
1042 in each word, which are upper case.

1043 **9.6 Functions That Produce Side Effects**

1044 Most functions are executed to obtain the **results** they produce but some
1045 functions are executed in order to **have them perform work that is not in the**
1046 **form of a result**. Functions that perform work that is not in the form of a result
1047 are said to produce **side effects**. Side effects include many forms of work such
1048 as sending information to the user, opening files, and changing values in the
1049 computer's memory.

1050 When a function produces a side effect which sends information to the user, this
1051 information has the words **Side Effects:** placed before it in the output instead of
1052 the word **Result:**. The **Echo()** and **Write()** functions are examples of functions
1053 that produce side effects and they are covered in the next section.

1054 **9.6.1 Printing Related Functions: Echo(), Write(), And Newline()**

1055 The printing related functions send text information to the user and this is
1056 usually referred to as "printing" in this document. However, it may also be called
1057 "echoing" and "writing".

1058 **9.6.1.1 Echo()**

1059 The **Echo()** function takes one expression (or multiple expressions separated by
1060 commas) evaluates each expression, and then prints the results as side effect
1061 output. The following examples illustrate this:

```
1062 In> Echo(1)
1063 Result> True
1064 Side Effects>
1065 1
```

1066 In this example, the number 1 was passed to the Echo() function, the number
1067 was evaluated (all numbers evaluate to themselves), and the result of the
1068 evaluation was then printed as a side effect. Notice that Echo() **also returned a**
1069 **result**. In MathPiper, all functions return a result, but functions whose main
1070 purpose is to produce a side effect usually just return a result of **True** if the side
1071 effect succeeded or **False** if it failed. In this case, Echo() returned a result of
1072 **True** because it was able to successfully print a 1 as its side effect.

1073 The next example shows multiple expressions being sent to Echo() (notice that
1074 the expressions are separated by commas):

```
1075 In> Echo(1,1+2,2*3)
1076 Result> True
1077 Side Effects>
1078 1 3 6
```

1079 The expressions were each evaluated and their results were returned (separated
1080 by spaces) as side effect output. If it is desired that commas be printed between
1081 the numbers in the output, simply place three commas between the expressions
1082 that are passed to Echo():

```
1083 In> Echo(1,,,1+2,,,2*3)
1084 Result> True
1085 Side Effects>
1086 1 , 3 , 6
```

1087 Each time an Echo() function is executed, it always forces the display to drop
1088 down to the next line after it is finished. This can be seen in the following
1089 program which is similar to the previous one except it uses a separate Echo()
1090 function to display each expression:

```
1091 %mathpiper
1092 Echo(1);
1093 Echo(1+2);
1094 Echo(2*3);
1095 %/mathpiper
1096 %output,preserve="false"
1097 Result: True
1098
1099 Side Effects:
1100 1
1101 3
1102 6
1103 . %/output
```

1104 Notice how the 1, the 3, and the 6 are each on their own line.

1105 Now that we have seen how Echo() works, lets use it to do something useful. If
1106 more than one expression is evaluated in a %mathpiper fold, only the result from
1107 the last expression that was evaluated (which is usually the bottommost
1108 expression) is displayed:

```
1109 %mathpiper
1110 a := 1;
1111 b := 2;
1112 c := 3;
1113 %/mathpiper
```

```
1114     %output,preserve="false"
1115     Result: 3
1116 .    %/output
```

1117 In MathPiper, programs are executed one line at a time, starting at the topmost
1118 line of code and working downwards from there. In this example, the line `a := 1;`
1119 is executed first, then the line `b := 2;` is executed, and so on. Notice, however,
1120 that even though we wanted to see what was in all three variables, only the
1121 content of the last variable was displayed.

1122 The following example shows how `Echo()` can be used to display the contents of
1123 all three variables:

```
1124 %mathpiper
1125 a := 1;
1126 Echo(a);
1127 b := 2;
1128 Echo(b);
1129 c := 3;
1130 Echo(c);
1131 %/mathpiper
1132     %output,preserve="false"
1133     Result: True
1134
1135     Side Effects:
1136     1
1137     2
1138     3
1139 .    %/output
```

1140 9.6.1.2 Echo Functions Are Useful For "Debugging" Programs

1141 The errors that are in a program are often called "bugs". This name came from
1142 the days when computers were the size of large rooms and were made using
1143 electromechanical parts. Periodically, bugs would crawl into the machines and
1144 interfere with its moving mechanical parts and this would cause the machine to
1145 malfunction. The bugs needed to be located and removed before the machine
1146 would run properly again.

1147 Of course, even back then most program errors were produced by programmers
1148 entering wrong programs or entering programs wrong, but they liked to say that
1149 all of the errors were caused by bugs and not by themselves! The process of
1150 fixing errors in a program became known as **debugging** and the names "bugs"

1151 and "debugging" are still used by programmers today.

1152 One of the standard ways to locate bugs in a program is to place **Echo()** function
1153 calls in the code at strategic places which **print the contents of variables and**
1154 **display messages**. These Echo() functions will enable you to see what your
1155 program is doing while it is running. After you have found and fixed the bugs in
1156 your program, you can remove the debugging Echo() function calls or comment
1157 them out if you think they may be needed later.

1158 9.6.1.3 Write()

1159 The **Write()** function is similar to the Echo() function except it does not
1160 automatically drop the display down to the next line after it finishes executing:

```
1161 %mathpiper
1162 Write(1);
1163 Write(1+2);
1164 Echo(2*3);
1165 %/mathpiper
1166     %output,preserve="false"
1167     Result: True
1168
1169     Side Effects:
1170     1 3 6
1171 .    %/output
```

1172 Write() and Echo() have other differences besides the one discussed here and
1173 more information about them can be found in the documentation for these
1174 functions.

1175 9.6.1.4 NewLine()

1176 The **NewLine()** function simply prints a blank line in the side effects output. It
1177 is useful for placing vertical space between printed lines:

```
1178 %mathpiper
1179 a := 1;
1180 Echo(a);
1181 NewLine();
1182 b := 2;
1183 Echo(b);
```



```
1184 NewLine();
1185 c := 3;
1186 Echo(c);

1187 %/mathpiper

1188     %output,preserve="false"
1189     Result: True
1190
1191     Side Effects:
1192     1
1193
1194     2
1195
1196     3
1197 . %/output
```

1196 9.7 Expressions Are Separated By Semicolons

1197 As discussed earlier, all of the expressions that are inside of a **%mathpiper** fold
1198 must have a semicolon (;) after them. However, the expressions executed in the
1199 **MathPiper console** did not have a semicolon after them. MathPiper actually
1200 requires that all expressions end with a semicolon, but one does not need to add
1201 a semicolon to an expression which is typed into the MathPiper console **because**
1202 **the console adds it automatically** when the expression is executed.

1203 9.7.1 Placing More Than One Expression On A Line In A Fold

1204 All the previous code examples have had each of their expressions on a separate
1205 line, but multiple expressions can also be placed on a single line because the
1206 semicolons tell MathPiper where one expression ends and the next one begins:

```
1207 %/mathpiper

1208 a := 1; Echo(a); b := 2; Echo(b); c := 3; Echo(c);

1209 %/mathpiper

1210     %output,preserve="false"
1211     Result: True
1212
1213     Side Effects:
1214     1
1215     2
1216     3
1217 . %/output
```

1218 The spaces that are in the code of this example are used to make the code more
1219 readable. Any spaces that are present within any expressions or between them
1220 are ignored by MathPiper and if we remove the spaces from the previous code,
1221 the output remains the same:

```
1222 %mathpiper
1223 a:=1;Echo(a);b:=2;Echo(b);c:= 3;Echo(c);
1224 %/mathpiper
1225     %output,preserve="false"
1226     Result: True
1227
1228     Side Effects:
1229     1
1230     2
1231     3
1232 .    %/output
```

1233 9.7.2 Placing Multiple Expressions In A Code Block

1234 A **code block** (which is also called a **compound expression**) consists of one or
1235 more expressions which are separated by semicolons and placed within an open
1236 bracket (**[**) and close bracket (**]**) pair. When a code block is evaluated, each
1237 expression in the block will be executed from left to right. The following
1238 example shows expressions being executed within of a code block inside the
1239 MathPiper console:

```
1240 In> [a := 1; Echo(a); b := 2; Echo(b); c := 3; Echo(c);]
1241 Result> True
1242 Side Effects>
1243 1
1244 2
1245 3
```

1246 Notice that all of the expressions were executed and 1-3 was printed as a side
1247 effect. Code blocks **always return the result of the last expression executed**
1248 **as the result of the whole block**. In this case, **True** was returned as the result
1249 because the last **Echo(c)** function returned **True**. If we place **another**
1250 **expression after the Echo(c) function**, however, **the block will execute this**
1251 **new expression last and its result will be the one returned by the block**:

```
1252 In> [a := 1; Echo(a); b := 2; Echo(b); c := 3; Echo(c); 2 + 2;]
1253 Result> 4
1254 Side Effects>
1255 1
```

1256 2
1257 3

1258 Finally, code blocks can have their contents placed on separate lines if desired:

```
1259 %mathpiper
1260 [
1261     a := 1;
1262
1263     Echo(a);
1264
1265     b := 2;
1266
1267     Echo(b);
1268
1269     c := 3;
1270
1271     Echo(c);
1272 ];
1273
1274 %/mathpiper
1275
1276     %output,preserve="false"
1277     Result: True
1278
1279     Side Effects:
1280     1
1281     2
1282     3
1283 .    %/output
```

1282 Code blocks are very powerful and we will be discussing them further in later
1283 sections.

1284 9.7.2.1 Automatic Bracket, Parentheses, And Brace Match Indicating

1285 In programming, most open brackets '[' have a close bracket ']', most open
1286 parentheses '(' have a close parentheses ')', and most open braces '{' have a
1287 close brace '}'. It is often difficult to make sure that each "open" character has a
1288 matching "close" character and if any of these characters don't have a match,
1289 then an error will be produced.

1290 Thankfully, most programming text editors have a character match indicating
1291 tool that will help locate problems. To try this tool, paste the following code into
1292 a .mrw file and following the directions that are present in its comments:

```
1293 %mathpiper
1294 /*
```

```
1295      Copy this code into a .mrw file.  Then, place the cursor
1296      to the immediate right of any {, }, [, ], (, or ) character.
1297      You should notice that the match to this character is
1298      indicated by a rectangle being drawing around it.
1299  */
```

```
1300  list := {1,2,3};
1301  [
1302      Echo("Hello");
1303      Echo(list);
1304  ];
1305  %/mathpiper
```

1306 9.8 Strings

1307 A **string** is a **value** that is used to hold text-based information. The typical
1308 expression that is used to create a string consists of **text which is enclosed**
1309 **within double quotes**. Strings can be assigned to variables just like numbers
1310 can and strings can also be displayed using the Echo() function. The following
1311 program assigns a string value to the variable 'a' and then echos it to the user:

```
1312 %mathpiper
1313 a := "Hello, I am a string.";
1314 Echo(a);
1315 %/mathpiper
1316 %output,preserve="false"
1317 Result: True
1318
1319 Side Effects:
1320 Hello, I am a string.
1321 . %/output
```

1322 9.8.1 The MathPiper Console and MathPiper Folds Can Access The Same 1323 Variables

1324 A useful aspect of using MathPiper inside of MathRider is that variables that are
1325 assigned inside of a **%mathpiper fold** are accessible inside of the **MathPiper**
1326 **console** and variables that are assigned inside of the **MathPiper console** are
1327 available inside of **%mathpiper folds**. For example, after the above fold is
1328 executed, the string that has been bound to variable 'a' can be displayed in the
1329 MathPiper console:

```
1330 In> a
1331 Result> "Hello, I am a string."
```

1332 9.8.2 Using Strings To Make Echo's Output Easier To Read

1333 When the Echo() function is used to print the values of multiple variables, it is
1334 often helpful to print some information next to each variable so that it is easier to
1335 determine which value came from which Echo() function in the code. The
1336 following program prints the name of the variable that each value came from
1337 next to it in the side effects output:

```
1338 %mathpiper
1339 a := 1;
1340 Echo("Variable a: ", a);
1341 b := 2;
1342 Echo("Variable b: ", b);
1343 c := 3;
1344 Echo("Variable c: ", c);
1345 %/mathpiper
1346     %output,preserve="false"
1347     Result: True
1348
1349     Side Effects:
1350     Variable a: 1
1351     Variable b: 2
1352     Variable c: 3
1353 .    %/output
```

1354 9.8.2.1 Combining Strings With The : Operator

1355 If you need to combine two or more strings into one string, you can use the :
1356 operator like this:

```
1357 In> "A" : "B" : "C"
1358 Result: "ABC"
1359 In> "Hello " : "there!"
1360 Result: "Hello there!"
```

1361 9.8.2.2 WriteString()

1362 The **WriteString()** function prints a string without shows the double quotes that

1363 are around it.. For example, here is the Write() function being used to print the
1364 string "Hello":

```
1365 In> Write("Hello")
1366 Result: True
1367 Side Effects:
1368 "Hello"
```

1369 Notice the double quotes? Here is how the WriteString() function prints "Hello":

```
1370 In> WriteString("Hello")
1371 Result: True
1372 Side Effects:
1373 Hello
```

1374 **9.8.2.3 NI()**

1375 The **NI()** (New Line) function is used with the : function to place newline
1376 characters inside of strings:

```
1377 In> WriteString("A": NI() : "B")
1378 Result: True
1379 Side Effects:
1380 A
1381 B
```

1382 **9.8.2.4 Space()**

1383 The Space() function is used to add spaces to printed output:

```
1384 In> WriteString("A"); Space(5); WriteString("B")
1385 Result: True
1386 Side Effects:
1387 A      B
```

```
1388 In> WriteString("A"); Space(10); WriteString("B")
1389 Result: True
1390 Side Effects:
1391 A          B
```

```
1392 In> WriteString("A"); Space(20); WriteString("B")
1393 Result: True
1394 Side Effects:
1395 A                      B
```

1396 **9.8.3 Accessing The Individual Letters In A String**

1397 Individual letters in a string (which are also called **characters**) can be accessed
1398 by placing the character's position number (also called an **index**) inside of

1399 brackets **[]** after the variable it is bound to. A character's position is determined
1400 by its distance from the left side of the string starting at 1. For example, in the
1401 string "Hello", 'H' is at position 1, 'e' is at position 2, etc. The following code
1402 shows individual characters in the above string being accessed:

```
1403 In> a := "Hello, I am a string."  
1404 Result> "Hello, I am a string."
```

```
1405 In> a[1]  
1406 Result> "H"
```

```
1407 In> a[2]  
1408 Result> "e"
```

```
1409 In> a[3]  
1410 Result> "l"
```

```
1411 In> a[4]  
1412 Result> "l"
```

```
1413 In> a[5]  
1414 Result> "o"
```

1415 **9.8.3.1 Indexing Before The Beginning Of A String Or Past The End Of A String**

1416 Lets see what happens if an index is used that is less than **1** or greater than the
1417 length of a given string. First, we will bind the string "Hello" to the variable 'a':

```
1418 In> a := "Hello"  
1419 Result: "Hello"
```

1420 Then, we'll index the character at position **1** and then the character at position **0**:

```
1421 In> a[1]  
1422 Result: "H"
```

```
1423 In> a[0]  
1424 Result:  
1425 Exception: In function "StringMidGet" :  
1426 bad argument number 1(counting from 1) :  
1427 The offending argument aindex evaluated to 0
```

1428 Notice that using an index of **0** resulted in an error.

1429 Next, lets access the character at position **5** (which is the 'o'), then the character
1430 at position **6** and finally the character at position **7**:

```
1431 In> a[5]
```

1432 `Result: "o"`

1433 `In> a[6]`

1434 `Result: ""`

1435 `In> a[7]`

1436 `Result:`

1437 `Exception: String index out of range: 8`

1438 The 'o' at position **5** was returned correctly, but accessing position **6** returned a
1439 double quote character (") and accessing position 7 resulted in an error. What
1440 you can see in this section is that errors are usually produced if an index is not
1441 set to the position of an actual character in a string.

1442 **9.9 Comments**

1443 Source code can often be difficult to understand and therefore all programming
1444 languages provide the ability for **comments** to be included in the code.

1445 Comments are used to explain what the code near them is doing and they are
1446 usually meant to be read by humans instead of being processed by a computer.
1447 Therefore, comments are ignored by the computer when a program is executed.

1448 There are two ways that MathPiper allows comments to be added to source code.
1449 The first way is by placing two forward slashes // to the left of any text that is
1450 meant to serve as a comment. The text from the slashes to the end of the line
1451 the slashes are on will be treated as a comment. Here is a program that contains
1452 comments which use slashes:

1453 `%mathpiper`

1454 `//This is a comment.`

1455 `x := 2; //Set the variable x equal to 2.`

1456 `%/mathpiper`

1457 `%output,preserve="false"`

1458 `Result: 2`

1459 `. %/output`

1460 When this program is executed, any text that starts with slashes is ignored.

1461 The second way to add comments to a MathPiper program is by enclosing the
1462 comments inside of slash-asterisk/asterisk-slash symbols /* */. This option is
1463 useful when a comment is too large to fit on one line. Any text between these
1464 symbols is ignored by the computer. This program shows a longer comment
1465 which has been placed between these symbols:


```
1466 %mathpiper
1467 /*
1468  This is a longer comment and it uses
1469  more than one line. The following
1470  code assigns the number 3 to variable
1471  x and then returns it as a result.
1472 */
1473 x := 3;
1474 %/mathpiper
1475     %output,preserve="false"
1476     Result: 3
1477 .    %/output
```

1478 **9.10 How To Tell If MathPiper Has Crashed And What To Do If It Has**

1479 Sometimes code will be evaluated which has one or more unusual errors in it and
1480 the errors will cause MathPiper to "crash". Unfortunately, beginners are more
1481 likely to crash MathPiper than more experienced programmers are because a
1482 beginner's program is more likely to have errors in it. When MathPiper crashes,
1483 no harm is done but it will not work correctly after that. **The only way to
1484 recover from a MathPiper crash is to exit MathRider and then relaunch
1485 it.** All the information in your buffers will be saved and preserved **but the
1486 contents of the console will not be.** Be sure to copy the contents of the
1487 console into a buffer and then save it before restarting.

1488 The main way to tell if MathRider has crashed is that it will indicate that **there
1489 are errors in lines of code that are actually fine.** If you are receiving an
1490 error in code that looks okay to you, simply restarting MathRider may fix the
1491 problem. If you restart MathRider and the error is still present, this usually
1492 means that there really is an error in the code.

1493 **9.11 Exercises**

1494 For the following exercises, create a new MathRider worksheet file called
1495 **book_1_section_9_exercises_<your first name>_<your last name>.mrw.**
1496 **(Note: there are no spaces in this file name).** For example, John Smith's
1497 worksheet would be called:

1498 **book_1_section_9_exercises_john_smith.mrw.**

1499 After this worksheet has been created, place your answer for each exercise that
1500 requires a fold into its own fold in this worksheet. Place a title attribute in the
1501 start tag of each fold which indicates the exercise the fold contains the solution
1502 to. The folds you create should look similar to this one:

```
1503 %mathpiper,title="Exercise 1"
```

```
1504 //Sample fold.
```

```
1505 %/mathpiper
```

1506 If an exercise uses the MathPiper console instead of a fold, copy the work you
1507 did in the console into the worksheet so it can be saved.

1508 9.11.1 Exercise 1

1509 Carefully read all of section 9. Evaluate each one of the examples in
1510 section 9 in the MathPiper worksheet you created or in the MathPiper
1511 console and verify that the results match the ones in the book. Copy all
1512 of the console examples you evaluated into your worksheet so they will be
1513 saved but do not put them in a fold.

1514 9.11.2 Exercise 2

1515 Change the precedence of the following expression using parentheses so that
1516 it prints 20 instead of 14:

```
1517 2 + 3 * 4
```

1518 9.11.3 Exercise 3

1519 Place the following calculations into a fold and then use one Echo()
1520 function per variable to print the results of the calculations. Put
1521 strings in the Echo() functions which indicate which variable each
1522 calculated value is bound to:

```
1523 a := 1+2+3+4+5;
```

```
1524 b := 1-2-3-4-5;
```

```
1525 c := 1*2*3*4*5;
```

```
1526 d := 1/2/3/4/5;
```

1527 9.11.4 Exercise 4

1528 Place the following calculations into a fold and then use one Echo()
1529 function to print the results of all the calculations on a single line
1530 (Remember, the Echo() function can print multiple values if they are
1531 separated by commas.):

```
1532 Clear(x);
```

```
1533 a := 2*2*2*2*2;
```

```
1534 b := 2^5;
```

```
1535 c := x^2 * x^3;
```

```
1536 d := 2^2 * 2^3;
```

1537 **9.11.5 Exercise 5**

1538 The following code assigns a string which contains all of the upper case
1539 letters of the alphabet to the variable **upper**. Each of the three Echo()
1540 functions prints an index number and the letter that is at that position in
1541 the string. Place this code into a fold and then continue the Echo()
1542 functions so that all 26 letters and their index numbers are printed

```
1543 upper := "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

```
1544 Echo(1,upper[1]);
```

```
1545 Echo(2,upper[2]);
```

```
1546 Echo(3,upper[3]);
```

1547 **9.11.6 Exercise 6**

1548 Use Echo() functions to print an index number and the character at this
1549 position for the following string (this is similar to what was done in
1550 Exercise 4.):

```
1551 extra := "!.@#$$%^&*() _+<>,?/{ }[]|\-= ";
```

```
1552 Echo(1,extra[1]);
```

```
1553 Echo(2,extra[2]);
```

```
1554 Echo(3,extra[3]);
```

1555 **9.11.7 Exercise 7**

1556 The following program uses strings and index numbers to print a person's
1557 name. Create a program which uses the three strings from this program to
1558 print the names of three of your favorite movie actors.

```
1559 %mathpiper
```

```
1560 /*
```

```
1561     This program uses strings and index numbers to print
```

```
1562     a person's name.
```

```
1563 */
```

```
1564 upper := "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

```
1565 lower := "abcdefghijklmnopqrstuvwxyz";
```

```
1566 extra := "!.@#$$%^&*() _+<>,?/{ }[]|\-= ";
```

```
1567 //Print "Mary Smith."
```

```
1568 Echo(upper[13],lower[1],lower[18],lower[25],extra[12],upper[19],lower[13],l
```

```
1569 ower[9],lower[20],lower[8],extra[1]);
```

```
1570 %/mathpiper
```

```
1571     %output,preserve="false"
```

```
1572         Result: True
1573
1574         Side Effects:
1575         Mary Smith.
1576     .    %/output
```

1577 10 Rectangular Selection Mode And Text Area Splitting

1578 10.1 Rectangular Selection Mode

1579 One capability that MathRider has that a word processor may not have is the
1580 ability to select rectangular sections of text. To see how this works, do the
1581 following:

- 1582 1) Type three or four lines of text into a text area.
- 1583 2) Hold down the **<Alt>** key then slowly press the **backslash key** (\) a few
1584 times. The bottom of the MathRider window contains a text field which
1585 MathRider uses to communicate information to the user. As **<Alt>** is
1586 repeatedly pressed, messages are displayed which read **Rectangular**
1587 **selection is on** and **Rectangular selection is off**.
- 1588 3) Turn rectangular selection on and then select some text in order to see
1589 how this is different than normal selection mode. **When you are done**
1590 **experimenting, set rectangular selection mode to off.**

1591 Most of the time normal selection mode is what you want to use but in certain
1592 situations rectangular selection mode is better.

1593 10.2 Text area splitting

1594 Sometimes it is useful to have two or more text areas open for a single document
1595 or multiple documents so that different parts of the documents can be edited at
1596 the same time. A situation where this would have been helpful was in the
1597 previous section where the output from an exercise in a MathRider worksheet
1598 contained a list of index numbers and letters which was useful for completing a
1599 later exercise.

1600 MathRider has this ability and it is called **splitting**. If you look just to the right
1601 of the toolbar there is an icon which looks like a blank window, an icon to the
1602 right of it which looks like a window which was split horizontally, and an icon to
1603 the right of the horizontal one which is split vertically. If you let your mouse
1604 hover over these icons, a short description will be displayed for each of them.

1605 Select a text area and then experiment with splitting it by pressing the horizontal
1606 and vertical splitting buttons. Move around these split text areas with their
1607 scroll bars and when you want to unsplit the document, just press the "**Unsplit**
1608 **All**" icon.

1609 10.3 Exercises

1610 For the following exercises, create a new MathRider worksheet file called
1611 **book_1_section_10_exercises_<your first name>_<your last name>.mrw**.

1612 (**Note: there are no spaces in this file name**). For example, John Smith's
1613 worksheet would be called:

1614 **book_1_section_10_exercises_john_smith.mrw.**

1615 For the following exercises, simply type your answers anywhere in the
1616 worksheet.

1617 **10.3.1 Exercise 1**

1618 Carefully read all of section 9 then answer the following questions:

1619 a) Give two examples where rectangular selection mode may be more useful
1620 than regular selection mode.

1621 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
```

```
1662 In> RandomInteger(100)
1663 Result> 82
1664 In> RandomInteger(100)
1665 Result> 93
1666 In> RandomInteger(100)
1667 Result> 32
```

1668 A range of random integers that does not start with 1 can also be generated by
1669 using the **two argument** version of **RandomInteger()**. For example, random
1670 integers between 25 and 75 can be obtained by passing RandomInteger() the
1671 lowest integer in the range and the highest one:

```
1672 In> RandomInteger(25, 75)
1673 Result: 28
1674 In> RandomInteger(25, 75)
1675 Result: 37
1676 In> RandomInteger(25, 75)
1677 Result: 58
1678 In> RandomInteger(25, 75)
1679 Result: 50
1680 In> RandomInteger(25, 75)
1681 Result: 70
```

1682 **11.2 Simulating The Rolling Of Dice**

1683 The following example shows the simulated rolling of a single six sided die using
1684 the RandomInteger() function:

```
1685 In> RandomInteger(6)
1686 Result> 5
1687 In> RandomInteger(6)
1688 Result> 6
1689 In> RandomInteger(6)
1690 Result> 3
1691 In> RandomInteger(6)
1692 Result> 2
1693 In> RandomInteger(6)
1694 Result> 5
```

1695 Code that simulates the rolling of two 6 sided dice can be evaluated in the
1696 MathPiper console by placing it within a **code block**. The following code
1697 outputs the sum of the two simulated dice:

```
1698 In> [a := RandomInteger(6); b := RandomInteger(6); a + b;]
1699 Result> 6
1700 In> [a := RandomInteger(6); b := RandomInteger(6); a + b;]
1701 Result> 12
1702 In> [a := RandomInteger(6); b := RandomInteger(6); a + b;]
1703 Result> 6
```



```
1704 In> [a := RandomInteger(6); b := RandomInteger(6); a + b;]
1705 Result> 4
1706 In> [a := RandomInteger(6); b := RandomInteger(6); a + b;]
1707 Result> 3
1708 In> [a := RandomInteger(6); b := RandomInteger(6); a + b;]
1709 Result> 8
```

1710 Now that we have the ability to simulate the rolling of two 6 sided dice, it would
1711 be interesting to determine if some sums of these dice occur more frequently
1712 than other sums. What we would like to do is to roll these simulated dice
1713 hundreds (or even thousands) of times and then analyze the sums that were
1714 produced. We don't have the programming capability to easily do this yet, but
1715 after we finish the section on **while loops**, we will.

1716 11.3 Exercises

1717 For the following exercises, create a new MathRider worksheet file called
1718 **book_1_section_11_exercises_<your first name>_<your last name>.mrw.**
1719 (**Note: there are no spaces in this file name**). For example, John Smith's
1720 worksheet would be called:

1721 **book_1_section_11_exercises_john_smith.mrw.**

1722 After this worksheet has been created, place your answer for each exercise that
1723 requires a fold into its own fold in this worksheet. Place a title attribute in the
1724 start tag of each fold which indicates the exercise the fold contains the solution
1725 to. The folds you create should look similar to this one:

```
1726 %mathpiper,title="Exercise 1"
```

```
1727 //Sample fold.
```

```
1728 %/mathpiper
```

1729 If an exercise uses the MathPiper console instead of a fold, copy the work you
1730 did in the console into the worksheet so it can be saved but do not put it in a fold.

1731 11.3.1 Exercise 1

1732 Carefully read all of section 11. Evaluate each one of the examples in
1733 section 11 in the MathPiper worksheet you created or in the MathPiper
1734 console and verify that the results match the ones in the book. Copy all
1735 of the console examples you evaluated into your worksheet so they will be
1736 saved but do not put them in a fold.

12 Making Decisions

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

12.1 Conditional Operators

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

Operator	Description
<code>x = y</code>	Returns 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
```

```
1757 In> 4 > 5
1758 Result> False
```

```
1759 In> 8 >= 8
1760 Result> True
```

```
1761 In> 5 <= 10
1762 Result> True
```

1763 The following examples show each of the conditional operators in Table 2 being
1764 used to compare values that have been assigned to variables **x** and **y**:

```
1765 %mathpiper
```

```
1766 // Example 1.
1767 x := 2;
1768 y := 3;
```

```
1769 Echo(x, "=", y, ":", x = y);
1770 Echo(x, "!= ", y, ":", x != y);
1771 Echo(x, "< ", y, ":", x < y);
1772 Echo(x, "<= ", y, ":", x <= y);
1773 Echo(x, "> ", y, ":", x > y);
1774 Echo(x, ">= ", y, ":", x >= y);
```

```
1775 %/mathpiper
```

```
1776 %output,preserve="false"
1777 Result: True
1778
1779 Side Effects:
1780 2 = 3 :False
1781 2 != 3 :True
1782 2 < 3 :True
1783 2 <= 3 :True
1784 2 > 3 :False
1785 2 >= 3 :False
1786 . %/output
```

```
1787 %mathpiper
```

```
1788 // Example 2.
1789 x := 2;
1790 y := 2;
```

```
1791 Echo(x, "=", y, ":", x = y);
1792 Echo(x, "!= ", y, ":", x != y);
1793 Echo(x, "< ", y, ":", x < y);
1794 Echo(x, "<= ", y, ":", x <= y);
1795 Echo(x, "> ", y, ":", x > y);
```

```
1796      Echo(x, ">= ", y, ":", x >= y);
```

```
1797  %/mathpiper
```

```
1798      %output,preserve="false"
```

```
1799      Result: True
```

```
1800
```

```
1801      Side Effects:
```

```
1802      2 = 2 :True
```

```
1803      2 != 2 :False
```

```
1804      2 < 2 :False
```

```
1805      2 <= 2 :True
```

```
1806      2 > 2 :False
```

```
1807      2 >= 2 :True
```

```
1808  .    %/output
```

```
1809  %mathpiper
```

```
1810  // Example 3.
```

```
1811  x := 3;
```

```
1812  y := 2;
```

```
1813  Echo(x, "=", y, ":", x = y);
```

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

```
1815  Echo(x, "< ", y, ":", x < y);
```

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

```
1817  Echo(x, "> ", y, ":", x > y);
```

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

```
1819  %/mathpiper
```

```
1820      %output,preserve="false"
```

```
1821      Result: True
```

```
1822
```

```
1823      Side Effects:
```

```
1824      3 = 2 :False
```

```
1825      3 != 2 :True
```

```
1826      3 < 2 :False
```

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

```
1828      3 > 2 :True
```

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

```
1830  .    %/output
```

```
1831  Conditional operators are placed at a lower level of precedence than the other
1832  operators we have covered to this point:
```

```
1833  ()    Parentheses are evaluated from the inside out.
```

```
1834  ^     Then exponents are evaluated right to left.
```

1835 *,%/, Then multiplication, remainder, and division operations are evaluated
1836 left to right.

1837 +, - Then addition and subtraction are evaluated left to right.

1838 =,!=,<,<=,>,>= Finally, conditional operators are evaluated.

1839 **12.2 Predicate Expressions**

1840 Expressions which return either **True** or **False** are called "**predicate**"
1841 expressions. By themselves, predicate expressions are not very useful and they
1842 only become so when they are used with special decision making functions, like
1843 the If() function (which is discussed in the next section).

1844 **12.3 Exercises**

1845 For the following exercises, create a new MathRider worksheet file called
1846 **book_1_section_12a_exercises_<your first name>_<your last name>.mrw.**
1847 (**Note: there are no spaces in this file name**). For example, John Smith's
1848 worksheet would be called:

1849 **book_1_section_12a_exercises_john_smith.mrw.**

1850 After this worksheet has been created, place your answer for each exercise that
1851 requires a fold into its own fold in this worksheet. Place a title attribute in the
1852 start tag of each fold which indicates the exercise the fold contains the solution
1853 to. The folds you create should look similar to this one:

1854 `%mathpiper,title="Exercise 1"`

1855 `//Sample fold.`

1856 `%/mathpiper`

1857 If an exercise uses the MathPiper console instead of a fold, copy the work you
1858 did in the console into the worksheet so it can be saved but do not put it in a fold.

1859 **12.3.1 Exercise 1**

1860 Carefully read all of section 12 up to this point. Evaluate each one of
1861 the examples in the sections you read in the MathPiper worksheet you
1862 created or in the MathPiper console and verify that the results match the
1863 ones in the book. Copy all of the console examples you evaluated into your
1864 worksheet so they will be saved but do not put them in a fold.

1865 **12.3.2 Exercise 2**

1866 Open a MathPiper session and evaluate the following predicate expressions:

1867 `In> 3 = 3`

1868 `In> 3 = 4`

1869 `In> 3 < 4`

1870 `In> 3 != 4`

1871 `In> -3 < 4`

1872 `In> 4 >= 4`

1873 `In> 1/2 < 1/4`

1874 `In> 15/23 < 122/189`

1875 `/*In the following two expressions, notice that 1/2 is not considered to be`
1876 `equal to .5 unless it is converted to a numerical value first.*/`

1877 `In> 1/2 = .5`

1878 `In> N(1/2) = .5`

1879 **12.3.3 Exercise 3**

1880 Come up with 10 predicate expressions of your own and evaluate them in the
1881 MathPiper console.

1882 **12.4 Making Decisions With The If() Function & Predicate Expressions**

1883 All programming languages have the ability to make decisions and the most
1884 commonly used function for making decisions in MathPiper is the **If()** function.

1885 There are two calling formats for the If() function:

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

1886 The way the first form of the If() function works is that it evaluates the first
1887 expression in its argument list (which is the "**predicate**" expression) and then
1888 looks at the value that is returned. If this value is **True**, the "**then**" expression
1889 that is listed second in the argument list is executed. If the predicate expression
1890 evaluates to **False**, the "**then**" expression is not executed. (Note: any function

1891 that accepts a predicate expression as a parameter can also accept the boolean
1892 values True and False).

1893 The following program uses an **If()** function to determine if the value in variable
1894 number is greater than 5. If number is greater than 5, the program will echo
1895 "Greater" and then "End of program":

```
1896 %mathpiper
1897 number := 6;
1898 If(number > 5, Echo(number, "is greater than 5.));
1899 Echo("End of program.");
1900 %/mathpiper
1901 %output,preserve="false"
1902 Result: True
1903
1904 Side Effects:
1905 6 is greater than 5.
1906 End of program.
1907 . %/output
```

1908 In this program, number has been set to 6 and therefore the expression number
1909 > 5 is **True**. When the **If()** function evaluates the **predicate expression** and
1910 determines it is **True**, it then executes the **first Echo()** function. The **second**
1911 **Echo()** function at the bottom of the program prints "End of program"
1912 regardless of what the If() function does. (**Note: semicolons cannot be placed**
1913 **after expressions which are in function calls.**)

1914 Here is the same program except that **number** has been set to **4** instead of **6**:

```
1915 %mathpiper
1916 number := 4;
1917 If(number > 5, Echo(number, "is greater than 5.));
1918 Echo("End of program.");
1919 %/mathpiper
1920 %output,preserve="false"
1921 Result: True
1922
1923 Side Effects:
1924 End of program.
1925 . %/output
```

1926 This time the expression **number > 4** returns a value of **False** which causes the
1927 **If()** function to not execute the "**then**" expression that was passed to it.

1928 12.4.1 If() Functions Which Include An "Else" Parameter

1929 The second form of the If() function takes a third "**else**" expression which is
1930 executed only if the predicate expression is **False**. This program is similar to the
1931 previous one except an "**else**" expression has been added to it:

```
1932 %mathpiper
1933 x := 4;
1934 If(x > 5, Echo(x, "is greater than 5."), Echo(x, "is NOT greater than 5.));
1935 Echo("End of program.");
1936 %/mathpiper
1937     %output, preserve="false"
1938     Result: True
1939
1940     Side Effects:
1941     4 is NOT greater than 5.
1942     End of program.
1943 .    %/output
```

1944 12.5 Exercises

1945 For the following exercises, create a new MathRider worksheet file called
1946 **book_1_section_12b_exercises_<your first name>_<your last name>.mrw**.
1947 (**Note: there are no spaces in this file name**). For example, John Smith's
1948 worksheet would be called:

1949 **book_1_section_12b_exercises_john_smith.mrw**.

1950 After this worksheet has been created, place your answer for each exercise that
1951 requires a fold into its own fold in this worksheet. Place a title attribute in the
1952 start tag of each fold which indicates the exercise the fold contains the solution
1953 to. The folds you create should look similar to this one:

```
1954 %mathpiper, title="Exercise 1"
1955 //Sample fold.
1956 %/mathpiper
```

1957 If an exercise uses the MathPiper console instead of a fold, copy the work you

1958 did in the console into the worksheet so it can be saved but do not put it in a fold.

1959 **12.5.1 Exercise 1**

1960 Carefully read all of section 12 starting at the end of the previous
1961 exercises and up to this point. Evaluate each one of the examples in the
1962 sections you read in the MathPiper worksheet you created or in the
1963 MathPiper console and verify that the results match the ones in the book.
1964 Copy all of the console examples you evaluated into your worksheet so they
1965 will be saved but do not put them in a fold.

1966 **12.5.2 Exercise 2**

1967 Write a program which uses the RandomInteger() function to simulate the
1968 flipping of a coin (Hint: you can use 1 to represent a head and 0 to
1969 represent a tail.). Use predicate expressions, the If() function, and the
1970 Echo() function to print the string "**The coin came up heads.**" or the string
1971 "**The coin came up tails.**", depending on what the simulated coin flip came
1972 up as when the code was executed.

1973 **12.6 The And(), Or(), & Not() Boolean Functions & Infix Notation**

1974 **12.6.1 And()**

1975 Sometimes a programmer needs to check if two or more expressions are all **True**
1976 and one way to do this is with the **And()** function. The And() function has **two**
1977 **calling formats** (or **notations**) and this is the first one:

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

1978 This calling format is able to accept one or more predicate expressions as input.
1979 If **all** of these expressions returns a value of **True**, the And() function will also
1980 return a **True**. However, if **any** of the expressions return a **False**, then the And()
1981 function will return a **False**. This can be seen in the following example:

```
1982 In> And(True, True)  
1983 Result> True
```

```
1984 In> And(True, False)  
1985 Result> False
```

```
1986 In> And(False, True)  
1987 Result> False
```

```
1988 In> And(True, True, True, True)  
1989 Result> True
```

```
1990 In> And(True, True, False, True)
1991 Result> False
```

1992 The second format (or notation) that can be used to call the And() function is
1993 called **infix** notation:

```
expression1 And expression2
```

1994 With **infix** notation, an expression is placed on both sides of the And() function
1995 name instead of being placed inside of parentheses that are next to it:

```
1996 In> True And True
1997 Result> True
```

```
1998 In> True And False
1999 Result> False
```

```
2000 In> False And True
2001 Result> False
```

2002 Infix notation can only accept **two** expressions at a time, but it is often more
2003 convenient to use than function calling notation. The following program also
2004 demonstrates the infix version of the And() function being used:

```
2005 %mathpiper
```

```
2006 a := 7;
2007 b := 9;
```

```
2008 Echo("1: ", a < 5 And b < 10);
2009 Echo("2: ", a > 5 And b > 10);
2010 Echo("3: ", a < 5 And b > 10);
2011 Echo("4: ", a > 5 And b < 10);
```

```
2012 If(a > 5 And b < 10, Echo("These expressions are both true."));
```

```
2013 %/mathpiper
```

```
2014 %output,preserve="false"
2015 Result: True
2016
2017 Side Effects:
2018 1: False
2019 2: False
2020 3: False
2021 4: True
2022 These expressions are both true.
2023 . %/output
```

2024 **12.6.2 Or()**

2025 The Or() function is similar to the And() function in that it has both a function
2026 calling format and an infix calling format and it only works with predicate
2027 expressions. However, instead of requiring that all expressions be **True** in order
2028 to return a **True**, Or() will return a **True** if **one or more expressions are True**.

2029 Here is the function calling format for Or():

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

2030 and this example shows Or() being used with function calling format:

2031 In> Or(True, False)

2032 Result> True

2033 In> Or(False, True)

2034 Result> True

2035 In> Or(False, False)

2036 Result> False

2037 In> Or(False, False, False, False)

2038 Result> False

2039 In> Or(False, True, False, False)

2040 Result> True

2041 The infix notation format for Or() is as follows:

```
expression1 Or expression2
```

2042 and this example shows infix notation being used:

2043 In> True Or False

2044 Result> True

2045 In> False Or True

2046 Result> True

2047 In> False Or False

2048 Result> False

2049 The following program also demonstrates the infix version of the Or() function
2050 being used:

```
2051 %mathpiper
2052 a := 7;
2053 b := 9;
2054 Echo("1: ", a < 5 Or b < 10);
2055 Echo("2: ", a > 5 Or b > 10);
2056 Echo("3: ", a > 5 Or b < 10);
2057 Echo("4: ", a < 5 Or b > 10);
2058 If(a < 5 Or b < 10, Echo("At least one of these expressions is true.));
2059 %/mathpiper
2060 %output,preserve="false"
2061 Result: True
2062
2063 Side Effects:
2064 1: True
2065 2: True
2066 3: True
2067 4: False
2068 At least one of these expressions is true.
2069 . %/output
```

2070 12.6.3 Not() & Prefix Notation

2071 The **Not()** function works with predicate expressions like the And() and Or()
2072 functions do, except it can only accept **one** expression as input. The way Not()
2073 works is that it changes a **True** value to a **False** value and a **False** value to a
2074 **True** value. Here is the Not()'s function calling format:

```
Not(expression)
```

2075 and this example shows Not() being used with function calling format:

```
2076 In> Not(True)
2077 Result> False
2078 In> Not(False)
2079 Result> True
```

2080 Instead of providing an alternative infix calling format like And() and Or() do,
2081 Not()'s second calling format uses **prefix** notation:

```
Not expression
```

2082 Prefix notation looks similar to function notation except no parentheses are used:

```
2083 In> Not True
2084 Result> False
```

```
2085 In> Not False
2086 Result> True
```

2087 Finally, here is a program that also uses the prefix version of Not():

```
2088 %mathpiper
2089 Echo("3 = 3 is ", 3 = 3);
2090 Echo("Not 3 = 3 is ", Not 3 = 3);
2091 %/mathpiper
2092     %output,preserve="false"
2093     Result: True
2094
2095     Side Effects:
2096     3 = 3 is True
2097     Not 3 = 3 is False
2098 .    %/output
```

2099 12.7 Exercises

2100 For the following exercises, create a new MathRider worksheet file called
2101 **book_1_section_12c_exercises_<your first name>_<your last name>.mrw.**
2102 (**Note: there are no spaces in this file name**). For example, John Smith's
2103 worksheet would be called:

2104 **book_1_section_12c_exercises_john_smith.mrw.**

2105 After this worksheet has been created, place your answer for each exercise that
2106 requires a fold into its own fold in this worksheet. Place a title attribute in the
2107 start tag of each fold which indicates the exercise the fold contains the solution
2108 to. The folds you create should look similar to this one:

```
2109 %mathpiper,title="Exercise 1"
2110 //Sample fold.
2111 %/mathpiper
```

2112 If an exercise uses the MathPiper console instead of a fold, copy the work you

2113 did in the console into the worksheet so it can be saved but do not put it in a fold.

2114 **12.7.1 Exercise 1**

2115 Carefully read all of section 12 starting at the end of the previous
2116 exercises and up to this point. Evaluate each one of the examples in the
2117 sections you read in the MathPiper worksheet you created or in the
2118 MathPiper console and verify that the results match the ones in the book.
2119 Copy all of the console examples you evaluated into your worksheet so they
2120 will be saved but do not put them in a fold.

2121 **12.7.2 Exercise 2**

2122 The following program simulates the rolling of two dice and prints a
2123 message if **both** of the two dice come up less than or equal to 3. Create a
2124 similar program which simulates the flipping of two coins and print the
2125 message "Both coins came up heads." if both coins come up heads.

```
2126 %mathpiper
2127 /*
2128     This program simulates the rolling of two dice and prints a message if
2129     both of the two dice come up less than or equal to 3.
2130 */
```

```
2131 die1 := RandomInteger(6);
2132 die2 := RandomInteger(6);

2133 Echo("Die1: ", die1, "   Die2: ", die2);
2134 NewLine();
```

```
2135 If( die1 <= 3 And die2 <= 3, Echo("Both dice came up <= to 3.") );
```

```
2136 %/mathpiper
```

2137 **12.7.3 Exercise 3**

2138 The following program simulates the rolling of two dice and prints a
2139 message if **either** of the two dice come up less than or equal to 3. Create
2140 a similar program which simulates the flipping of two coins and print the
2141 message "At least one coin came up heads." if at least one coin comes up
2142 heads.

```
2143 %mathpiper
2144 /*
2145     This program simulates the rolling of two dice and prints a message if
2146     either of the two dice come up less than or equal to 3.
2147 */
```

```
2148 die1 := RandomInteger(6);
2149 die2 := RandomInteger(6);
```

```
2150 Echo("Die1: ", die1, " Die2: ", die2);
2151 NewLine();

2152 If( die1 <= 3 Or die2 <= 3, Echo("At least one die came up <= 3.") );

2153 %/mathpiper
```

2154 **13 The While() Looping Function & Bodied Notation**

2155 Many kinds of machines, including computers, derive much of their power from
2156 the principle of **repeated cycling**. **Repeated cycling** in a MathPiper program
2157 means to execute one or more expressions over and over again and this process
2158 is called "**looping**". MathPiper provides a number of ways to implement **loops**
2159 in a program and these ways range from straight-forward to subtle.

2160 We will begin discussing looping in MathPiper by starting with the straight-
2161 forward **While** function. The calling format for the **While** function is as follows:

```
2162 While(predicate)  
2163 [  
2164     body_expressions  
2165 ];
```

2166 The **While** function is similar to the **If** function except it will repeatedly execute
2167 the expressions it contains as long as its "predicate" expression is **True**. As soon
2168 as the predicate expression returns a **False**, the While() function skips the
2169 expressions it contains and execution continues with the expression that
2170 immediately follows the While() function (if there is one).

2171 The expressions which are contained in a While() function are called its "**body**"
2172 and all functions which have body expressions are called "**bodied**" functions. If
2173 a body contains more than one expression then these expressions need to be
2174 placed within a **code block** (code blocks were discussed in an earlier section).
2175 What a function's body is will become clearer after studying some example
2176 programs.

2177 **13.1 Printing The Integers From 1 to 10**

2178 The following program uses a While() function to print the integers from 1 to 10:

```
2179 %mathpiper  
  
2180 // This program prints the integers from 1 to 10.  
  
2181 /*  
2182     Initialize the variable count to 1  
2183     outside of the While "loop".  
2184 */  
2185 count := 1;  
  
2186 While(count <= 10)  
2187 [  
2188     Echo(count);
```



```
2189
2190     count := count + 1; //Increment count by 1.
2191 ];
2192 %/mathpiper
2193     %output,preserve="false"
2194     Result: True
2195
2196     Side Effects:
2197     1
2198     2
2199     3
2200     4
2201     5
2202     6
2203     7
2204     8
2205     9
2206     10
2207 . %/output
```

2208 In this program, a single variable called **count** is created. It is used to tell the
2209 Echo() function which integer to print and it is also used in the predicate
2210 expression that determines if the While() function should continue to **loop** or not.

2211 When the program is executed, 1 is placed into **count** and then the While()
2212 function is called. The predicate expression **count** <= 10 becomes **1** <= 10
2213 and, since 1 is indeed less than or equal to 10, a value of **True** is returned by the
2214 predicate expression.

2215 The While() function sees that the predicate expression returned a **True** and
2216 therefore it executes all of the expressions inside of its **body** from top to bottom.

2217 The Echo() function prints the current contents of count (which is 1) and then the
2218 expression count := count + 1 is executed.

2219 The expression **count := count + 1** is a standard expression form that is used in
2220 many programming languages. Each time an expression in this form is
2221 evaluated, it **increases the variable it contains by 1**. Another way to describe
2222 the effect this expression has on **count** is to say that it **increments count by 1**.

2223 In this case **count** contains **1** and, after the expression is evaluated, **count**
2224 contains **2**.

2225 After the last expression inside the body of the While() function is executed, the
2226 While() function reevaluates its predicate expression to determine whether it
2227 should continue looping or not. Since **count** is **2** at this point, the predicate
2228 expression returns **True** and the code inside the body of the While() function is
2229 executed again. This loop will be repeated until **count** is incremented to **11** and
2230 the predicate expression returns **False**.

2231 **13.2 Printing The Integers From 1 to 100**

2232 The previous program can be adjusted in a number of ways to achieve different
2233 results. For example, the following program prints the integers from 1 to 100 by
2234 changing the **10** in the predicate expression to **100**. A Write() function is used in
2235 this program so that its output is displayed on the same line until it encounters
2236 the **wrap margin** in MathRider (which can be set in Utilities -> Buffer
2237 Options...).

```
2238 %mathpiper
2239 // Print the integers from 1 to 100.
2240 count := 1;
2241 While(count <= 100)
2242 [
2243     Write(count,,);
2244     count := count + 1; //Increment count by 1.
2245 ];
2246
2247 %/mathpiper
2248     %output,preserve="false"
2249     Result: True
2250
2251     Side Effects:
2252     1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,
2253     24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,
2254     44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,
2255     64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,
2256     84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100
2257 . %/output
```

2258 **13.3 Printing The Odd Integers From 1 To 99**

2259 The following program prints the odd integers from 1 to 99 by changing the
2260 **increment value** in the increment expression from **1** to **2**:

```
2261 %mathpiper
2262 //Print the odd integers from 1 to 99.
2263 x := 1;
2264 While(x <= 100)
2265 [
2266     Write(x,,);
```

```
2267     x := x + 2;    //Increment x by 2.
2268 ];
2269 %/mathpiper
2270     %output,preserve="false"
2271     Result: True
2272
2273     Side Effects:
2274     1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43,
2275     45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71, 73, 75, 77, 79, 81, 83,
2276     85, 87, 89, 91, 93, 95, 97, 99
2277 .    %/output
```

2278 **13.4 Printing The Integers From 1 To 100 In Reverse Order**

2279 Finally, the following program prints the integers from 1 to 100 in reverse order:

```
2280 %mathpiper
2281 //Print the integers from 1 to 100 in reverse order.
2282 x := 100;
2283 While(x >= 1)
2284 [
2285     Write(x, , );
2286     x := x - 1;    //Decrement x by 1.
2287 ];
2288 %/mathpiper
2289     %output,preserve="false"
2290     Result: True
2291
2292     Side Effects:
2293     100, 99, 98, 97, 96, 95, 94, 93, 92, 91, 90, 89, 88, 87, 86, 85, 84, 83, 82,
2294     81, 80, 79, 78, 77, 76, 75, 74, 73, 72, 71, 70, 69, 68, 67, 66, 65, 64, 63,
2295     62, 61, 60, 59, 58, 57, 56, 55, 54, 53, 52, 51, 50, 49, 48, 47, 46, 45, 44,
2296     43, 42, 41, 40, 39, 38, 37, 36, 35, 34, 33, 32, 31, 30, 29, 28, 27, 26, 25,
2297     24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4,
2298     3, 2, 1
2299 .    %/output
```

2300 In order to achieve the reverse ordering, this program had to initialize **x** to **100**,
2301 check to see if **x** was **greater than or equal to 1** ($x \geq 1$), and **decrement** **x** by
2302 **subtracting 1 from it** instead of adding 1 to it.

2303 **13.5 Expressions Inside Of Code Blocks Are Indented**

2304 In the programs in the previous sections which use while loops, notice that the
2305 expressions which are inside of the While() function's code block are **indented**.
2306 These expressions do not need to be indented to execute properly, but doing so
2307 makes the program easier to read.

2308 **13.6 Long-Running Loops, Infinite Loops, & Interrupting Execution**

2309 It is easy to create a loop that will execute a **large number of times**, or even **an**
2310 **infinite number of times**, either on purpose or by mistake. When you execute
2311 a program that contains an **infinite loop**, it will run until you tell MathPiper to
2312 **interrupt** its execution. This is done by opening the MathPiper console and then
2313 pressing the **"Stop"** button which it contains. The Stop button is circular and it
2314 has an X on it. (**Note: currently this button only works if MathPiper is**
2315 **executed inside of a %mathpiper fold.**)

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

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

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

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

2337 **13.7 A Program That Simulates Rolling Two Dice 50 Times**

2338 The following program is larger than the previous programs that have been
2339 discussed in this book, but it is also more interesting and more useful. It uses a
2340 While() loop to simulate the rolling of two dice 50 times and it records how many
2341 times each possible sum has been rolled so that this data can be printed. The
2342 comments in the code explain what each part of the program does. (Remember, if
2343 you copy this program to a MathRider worksheet, you can use **rectangular**
2344 **selection mode** to easily remove the line numbers).

```
2345 %mathpiper
2346 /*
2347     This program simulates rolling two dice 50 times.
2348 */
2349 /*
2350     These variables are used to record how many times
2351     a possible sum of two dice has been rolled. They are
2352     all initialized to 0 before the simulation begins.
2353 */
2354 numberOfTwosRolled := 0;
2355 numberOfThreesRolled := 0;
2356 numberOfFoursRolled := 0;
2357 numberOfFivesRolled := 0;
2358 numberOfSixesRolled := 0;
2359 numberOfSevensRolled := 0;
2360 numberOfEightsRolled := 0;
2361 numberOfNinesRolled := 0;
2362 numberOfTensRolled := 0;
2363 numberOfElevensRolled := 0;
2364 numberOfTwelvesRolled := 0;
2365 //This variable keeps track of the number of the current roll.
2366 roll := 1;
2367 Echo("These are the rolls:");
2368 /*
2369     The simulation is performed inside of this while loop. The number of
2370     times the dice will be rolled can be changed by changing the number 50
2371     which is in the While function's predicate expression.
2372 */
2373 While(roll <= 50)
2374 [
2375     //Roll the dice.
2376     die1 := RandomInteger(6);
2377     die2 := RandomInteger(6);
```

```
2378
2379
2380 //Calculate the sum of the two dice.
2381 rollSum := die1 + die2;
2382
2383
2384 /*
2385 Print the sum that was rolled. Note: if a large number of rolls
2386 is going to be performed (say > 1000), it would be best to comment
2387 out this Write() function so that it does not put too much text
2388 into the output fold.
2389 */
2390 Write(rollSum,,);
2391
2392
2393 /*
2394 These If() functions determine which sum was rolled and then add
2395 1 to the variable which is keeping track of the number of times
2396 that sum was rolled.
2397 */
2398 If(rollSum = 2, numberOfTwosRolled := numberOfTwosRolled + 1);
2399 If(rollSum = 3, numberOfThreesRolled := numberOfThreesRolled + 1);
2400 If(rollSum = 4, numberOfFoursRolled := numberOfFoursRolled + 1);
2401 If(rollSum = 5, numberOfFivesRolled := numberOfFivesRolled + 1);
2402 If(rollSum = 6, numberOfSixesRolled := numberOfSixesRolled + 1);
2403 If(rollSum = 7, numberOfSevensRolled := numberOfSevensRolled + 1);
2404 If(rollSum = 8, numberOfEightsRolled := numberOfEightsRolled + 1);
2405 If(rollSum = 9, numberOfNinesRolled := numberOfNinesRolled + 1);
2406 If(rollSum = 10, numberOfTensRolled := numberOfTensRolled + 1);
2407 If(rollSum = 11, numberOfElevensRolled := numberOfElevensRolled+1);
2408 If(rollSum = 12, numberOfTwelvesRolled := numberOfTwelvesRolled+1);
2409
2410
2411 //Increment the roll variable to the next roll number.
2412 roll := roll + 1;
2413 ];
2414
2415 //Print the contents of the sum count variables for visual analysis.
2416 NewLine();
2417 NewLine();
2418 Echo("Number of Twos rolled: ", numberOfTwosRolled);
2419 Echo("Number of Threes rolled: ", numberOfThreesRolled);
2420 Echo("Number of Fours rolled: ", numberOfFoursRolled);
2421 Echo("Number of Fives rolled: ", numberOfFivesRolled);
2422 Echo("Number of Sixes rolled: ", numberOfSixesRolled);
2423 Echo("Number of Sevens rolled: ", numberOfSevensRolled);
2424 Echo("Number of Eights rolled: ", numberOfEightsRolled);
2425 Echo("Number of Nines rolled: ", numberOfNinesRolled);
2426 Echo("Number of Tens rolled: ", numberOfTensRolled);
2427 Echo("Number of Elevens rolled: ", numberOfElevensRolled);
2428 Echo("Number of Twelves rolled: ", numberOfTwelvesRolled);
```

```
2428 %/mathpiper
2429     %output,preserve="false"
2430     Result: True
2431
2432     Side effects:
2433     These are the rolls:
2434     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,
2435     12,7,8,12,6,8,10,10,5,9,8,4,5,3,5,7,7,4,6,7,7,5,8,
2436
2437     Number of Twos rolled: 0
2438     Number of Threes rolled: 3
2439     Number of Fours rolled: 6
2440     Number of Fives rolled: 4
2441     Number of Sixes rolled: 6
2442     Number of Sevens rolled: 13
2443     Number of Eights rolled: 6
2444     Number of Nines rolled: 3
2445     Number of Tens rolled: 2
2446     Number of Elevens rolled: 4
2447     Number of Twelves rolled: 3
2448 . %/output
```

2449 13.8 Exercises

2450 For the following exercises, create a new MathRider worksheet file called
2451 **book_1_section_13_exercises_<your first name>_<your last name>.mrw.**
2452 **(Note: there are no spaces in this file name).** For example, John Smith's
2453 worksheet would be called:

2454 **book_1_section_13_exercises_john_smith.mrw.**

2455 After this worksheet has been created, place your answer for each exercise that
2456 requires a fold into its own fold in this worksheet. Place a title attribute in the
2457 start tag of each fold which indicates the exercise the fold contains the solution
2458 to. The folds you create should look similar to this one:

```
2459 %mathpiper,title="Exercise 1"
```

```
2460 //Sample fold.
```

```
2461 %/mathpiper
```

2462 If an exercise uses the MathPiper console instead of a fold, copy the work you
2463 did in the console into the worksheet so it can be saved but do not put it in a fold.

2464 13.8.1 Exercise 1

2465 Carefully read all of section 13 up to this point. Evaluate each one of
2466 the examples in the sections you read in the MathPiper worksheet you
2467 created or in the MathPiper console and verify that the results match the
2468 ones in the book. Copy all of the console examples you evaluated into your
2469 worksheet so they will be saved but do not put them in a fold.

2470 13.8.2 Exercise 2

2471 Create a program which uses a while loop to print the even integers from 2
2472 to 50 inclusive.

2473 13.8.3 Exercise 3

2474 Create a program which prints all the multiples of 5 between 5 and 50
2475 inclusive.

2476 13.8.4 Exercise 4

2477 Create a program which simulates the flipping of a coin 500 times. Print
2478 the number of times the coin came up heads and the number of times it came
2479 up tails after the loop is finished executing.

2480 14 Predicate Functions

2481 A **predicate function** is a function that either returns **True** or **False**. Most
2482 predicate functions in MathPiper have names which begin with "**Is**". For
2483 example, **IsEven()**, **IsOdd()**, **IsInteger()**, etc. The following examples show
2484 some of the predicate functions that are in MathPiper:

2485 In> IsEven(4)

2486 Result> True

2487 In> IsEven(5)

2488 Result> False

2489 In> IsZero(0)

2490 Result> True

2491 In> IsZero(1)

2492 Result> False

2493 In> IsNegativeInteger(-1)

2494 Result> True

2495 In> IsNegativeInteger(1)

2496 Result> False

2497 In> IsPrime(7)

2498 Result> True

2499 In> IsPrime(100)

2500 Result> False

2501 There is also an **IsBound()** and an **IsUnbound()** function that can be used to
2502 determine whether or not a value is bound to a given variable:

2503 In> a

2504 Result> a

2505 In> IsBound(a)

2506 Result> False

2507 In> a := 1

2508 Result> 1

2509 In> IsBound(a)

2510 Result> True

2511 In> Clear(a)

2512 Result> True

```
2513 In> a
2514 Result> a
```

```
2515 In> IsBound(a)
2516 Result> False
```

2517 The complete list of predicate functions is contained in the **User**
2518 **Functions/Predicates** node in the MathPiperDocs plugin.

2519 **14.1 Finding Prime Numbers With A Loop**

2520 Predicate functions are very powerful when they are combined with loops
2521 because they can be used to automatically make numerous checks. The
2522 following program uses a while loop to pass the integers 1 through 20 (one at a
2523 time) to the **IsPrime()** function in order to determine which integers are prime
2524 and which integers are not prime:

```
2525 %mathpiper
2526 //Determine which numbers between 1 and 20 (inclusive) are prime.
2527 x := 1;
2528 While(x <= 20)
2529 [
2530     primeStatus := IsPrime(x);
2531     Echo(x, "is prime: ", primeStatus);
2532     x := x + 1;
2533 ];
2534
2535 %/mathpiper
2536
2537 %output,preserve="false"
2538 Result: True
2539
2540 Side Effects:
2541 1 is prime: False
2542 2 is prime: True
2543 3 is prime: True
2544 4 is prime: False
2545 5 is prime: True
2546 6 is prime: False
2547 7 is prime: True
2548 8 is prime: False
2549 9 is prime: False
2550 10 is prime: False
2551 11 is prime: True
2552 12 is prime: False
```

```
2553         13 is prime: True
2554         14 is prime: False
2555         15 is prime: False
2556         16 is prime: False
2557         17 is prime: True
2558         18 is prime: False
2559         19 is prime: True
2560         20 is prime: False
2561     .    %/output
```

2562 This program worked fairly well, but it is limited because it prints a line for each
2563 prime number and also each non-prime number. This means that if large ranges
2564 of integers were processed, enormous amounts of output would be produced.
2565 The following program solves this problem by using an If() function to only print
2566 a number if it is prime:

```
2567 %mathpiper
2568 //Print the prime numbers between 1 and 50 (inclusive).
2569 x := 1;
2570 While(x <= 50)
2571 [
2572     primeStatus := IsPrime(x);
2573     If(primeStatus = True, Write(x,,) );
2574     x := x + 1;
2575 ]
2576
2577 ];
2578 %/mathpiper
2579     %output,preserve="false"
2580     Result: True
2581
2582     Side Effects:
2583     2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,
2584 .    %/output
```

2585 This program is able to process a much larger range of numbers than the
2586 previous one without having its output fill up the text area. However, the
2587 program itself can be shortened by moving the **IsPrime()** function **inside** of the
2588 **If()** function instead of using the **primeStatus** variable to communicate with it:

```
2589 %mathpiper
2590 /*
```

```
2591     Print the prime numbers between 1 and 50 (inclusive).
2592     This is a shorter version which places the IsPrime() function
2593     inside of the If() function instead of using a variable.
2594 */
2595 x := 1;
2596 While(x <= 50)
2597 [
2598     If(IsPrime(x), Write(x,,) );
2599     x := x + 1;
2601 ];
2602 %/mathpiper
2603     %output,preserve="false"
2604     Result: True
2605
2606     Side Effects:
2607     2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,
2608     . %/output
```

2609 **14.2 Finding The Length Of A String With The Length() Function**

2610 Strings can contain zero or more characters and the **Length()** function can be
2611 used to determine how many characters a string holds:

```
2612 In> s := "Red"
2613 Result> "Red"
```

```
2614 In> Length(s)
2615 Result> 3
```

2616 In this example, the string "Red" is assigned to the variable **s** and then **s** is
2617 passed to the **Length()** function. The **Length()** function returned a **3** which
2618 means the string contained **3 characters**.

2619 The following example shows that strings can also be passed to functions
2620 directly:

```
2621 In> Length("Red")
2622 Result> 3
```

2623 An **empty string** is represented by **two double quote marks with no space in**
2624 **between them**. The **length** of an empty string is **0**:

```
2625 In> Length("")
2626 Result> 0
```

2627 **14.3 Converting Numbers To Strings With The String() Function**

2628 Sometimes it is useful to convert a number to a string so that the individual
2629 digits in the number can be analyzed or manipulated. The following example
2630 shows a **number** being converted to a **string** with the **String()** function so that
2631 its **leftmost** and **rightmost** digits can be placed into **variables**:

```
2632 In> number := 523
2633 Result> 523
```

```
2634 In> stringNumber := String(number)
2635 Result> "523"
```

```
2636 In> leftmostDigit := stringNumber[1]
2637 Result> "5"
```

```
2638 In> rightmostDigit := stringNumber[ Length(stringNumber) ]
2639 Result> "3"
```

2640 Notice that the Length() function is used here to determine which character in
2641 **stringNumber** held the **rightmost** digit.

2642 **14.4 Finding Prime Numbers Which End With 7 (And Multi-line Function** 2643 **Calls)**

2644 Now that we have covered how to turn a number into a string, lets use this
2645 ability inside a loop. The following program finds all the **prime numbers**
2646 between **1** and **500** which have a **7 as their rightmost digit**. There are three
2647 important things which are shown in this program:

2648 1) Function calls **can have their parameters placed on more than one**
2649 **line** if the parameters are too long to fit on a **single line**. In this case, a long
2650 code block is being placed inside of an If() function.

2651 2) Code blocks (which are considered to be compound expressions) **cannot**
2652 **have a semicolon placed after them if they are in a function call**. If a
2653 semicolon is placed after this code block, an error will be produced.

2654 3) If() functions can be placed inside of other If() functions in order to make
2655 more complex decisions. This is referred to as **nesting** functions.

2656 When the program is executed, it finds 24 prime numbers which have 7 as their
2657 rightmost digit:

```
2658 %mathpiper
2659 /*
2660      Find all the prime numbers between 1 and 500 which have a 7
2661      as their rightmost digit.
2662 */
2663 x := 1;
2664 While(x <= 500)
2665 [
2666     //Notice how function parameters can be put on more than one line.
2667     If(IsPrime(x),
2668         [
2669             stringVersionOfNumber := String(x);
2670
2671             stringLength := Length(stringVersionOfNumber);
2672
2673             //Notice that If() functions can be placed inside of other
2674             // If() functions.
2675             If(stringVersionOfNumber[stringLength] = "7", Write(x,,) );
2676
2677         ] //Notice that semicolons cannot be placed after code blocks
2678         //which are in function calls.
2679     ); //This is the close parentheses for the outer If() function.
2680     x := x + 1;
2681 ];
2682
2684 %/mathpiper
2685     %output,preserve="false"
2686     Result: True
2687
2688     Side Effects:
2689     7,17,37,47,67,97,107,127,137,157,167,197,227,257,277,307,317,
2690     337,347,367,397,457,467,487,
2691     . %/output
```

2692 It would be nice if we had the ability to store these numbers someplace so that
2693 they could be processed further and this is discussed in the next section.

2694 14.5 Exercises

2695 For the following exercises, create a new MathRider worksheet file called
2696 **book_1_section_14_exercises_<your first name>_<your last name>.mrw.**
2697 (**Note: there are no spaces in this file name**). For example, John Smith's
2698 worksheet would be called:

2699 book_1_section_14_exercises_john_smith.mrw.

2700 After this worksheet has been created, place your answer for each exercise that
2701 requires a fold into its own fold in this worksheet. Place a title attribute in the
2702 start tag of each fold which indicates the exercise the fold contains the solution
2703 to. The folds you create should look similar to this one:

```
2704 %mathpiper,title="Exercise 1"
```

```
2705 //Sample fold.
```

```
2706 %/mathpiper
```

2707 If an exercise uses the MathPiper console instead of a fold, copy the work you
2708 did in the console into the worksheet so it can be saved but do not put it in a fold.

2709 14.5.1 Exercise 1

2710 Carefully read all of section 14 up to this point. Evaluate each one of
2711 the examples in the sections you read in the MathPiper worksheet you
2712 created or in the MathPiper console and verify that the results match the
2713 ones in the book. Copy all of the console examples you evaluated into your
2714 worksheet so they will be saved but do not put them in a fold.

2715 14.5.2 Exercise 2

2716 Write a program which uses a loop to determine how many prime numbers there
2717 are between 1 and 1000. You do not need to print the numbers themselves,
2718 just how many there are.

2719 14.5.3 Exercise 3

2720 Write a program which uses a loop to print all of the prime numbers between
2721 10 and 99 which contain the digit 3 in either their 1's place, or their
2722 10's place, or both places.

2723 15 Lists: Values That Hold Sequences Of Expressions

2724 The **list** value type is designed to hold expressions in an **ordered collection** or
2725 **sequence**. Lists are very flexible and they are one of the most heavily used
2726 value types in MathPiper. Lists can **hold expressions of any type**, they can be
2727 made to **grow and shrink as needed**, and they can be **nested**. Expressions in a
2728 list can be **accessed by their position** in the list (similar to the way that
2729 characters in a string are accessed) and they can also be **replaced by other**
2730 **expressions**.

2731 One way to create a list is by placing zero or more expressions separated by
2732 commas inside of a **pair of braces {}**. In the following example, a list is created
2733 that contains various expressions and then it is assigned to the variable **x**:

```
2734 In> x := {7,42,"Hello",1/2,var}  
2735 Result> {7,42,"Hello",1/2,var}
```

```
2736 In> x  
2737 Result> {7,42,"Hello",1/2,var}
```

2738 The number of expressions in a list can be determined with the **Length()**
2739 function:

```
2740 In> Length({7,42,"Hello",1/2,var})  
2741 Result> 5
```

2742 A single expression in a list can be accessed by placing a set of **brackets []** to
2743 the right of the variable that is bound to the list and then putting the
2744 expression's position number inside of the brackets (**Note: the first expression**
2745 **in the list is at position 1 counting from the left end of the list**):

```
2746 In> x[1]  
2747 Result> 7
```

```
2748 In> x[2]  
2749 Result> 42
```

```
2750 In> x[3]  
2751 Result> "Hello"
```

```
2752 In> x[4]  
2753 Result> 1/2
```

```
2754 In> x[5]  
2755 Result> var
```

2756 The **1st** and **2nd** expressions in this list are **integers**, the **3rd** expression is a

2757 **string**, the **4th** expression is a **rational number** and the **5th** expression is an
2758 **unbound variable**.

2759 Lists can also hold other lists as shown in the following example:

```
2760 In> x := {20, 30, {31, 32, 33}, 40}
```

```
2761 Result> {20,30,{31,32,33},40}
```

```
2762 In> x[1]
```

```
2763 Result> 20
```

```
2764 In> x[2]
```

```
2765 Result> 30
```

```
2766 In> x[3]
```

```
2767 Result> {31,32,33}
```

```
2768 In> x[4]
```

```
2769 Result> 40
```

```
2770
```

2771 The expression in the **3rd** position in the list is another **list** which contains the
2772 integers **31**, **32**, and **33**.

2773 An expression in this second list can be accessed by two **two sets of brackets**:

```
2774 In> x[3][2]
```

```
2775 Result> 32
```

2776 The **3** inside of the first set of brackets accesses the **3rd** member of the **first** list
2777 and the **2** inside of the second set of brackets accesses the **2nd** member of the
2778 **second** list.

2779 **15.1 Append() & Nondestructive List Operations**

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

2780 The **Append()** function adds an expression to the end of a list:

```
2781 In> testList := {21,22,23}
```

```
2782 Result> {21,22,23}
```

```
2783 In> Append(testList, 24)
```

```
2784 Result> {21,22,23,24}
```

2785 However, instead of changing the **original** list, **Append()** creates a **copy** of the
2786 **original** list and appends the expression to the **copy**. This can be confirmed by
2787 evaluating the variable **testList** after the **Append()** function has been called:

```
2788 In> testList
2789 Result> {21,22,23}
```

2790 Notice that the list that is bound to **testList** was not modified by the **Append()**
2791 function. This is called a **nondestructive list operation** and **most MathPiper**
2792 **functions that manipulate lists do so nondestructively**. To have the new list
2793 bound to the variable that is being used, the following technique can be
2794 employed:

```
2795 In> testList := {21,22,23}
2796 Result> {21,22,23}

2797 In> testList := Append(testList, 24)
2798 Result> {21,22,23,24}

2799 In> testList
2800 Result> {21,22,23,24}
```

2801 After this code has been executed, the new list has indeed been bound to
2802 **testList** as desired.

2803 There are some functions, such as **DestructiveAppend()**, which **do** change the
2804 original list and most of them begin with the word "Destructive". These are
2805 called "destructive functions" and they are advanced functions which are not
2806 covered in this book.

2807 **15.2 Using While Loops With Lists**

2808 Functions that loop can be used to **select each expression in a list in turn** so
2809 that an operation can be performed on these expressions. The following
2810 program uses a while loop to print each of the expressions in a list:

```
2811 %mathpiper
2812 //Print each number in the list.

2813 x := {55,93,40,21,7,24,15,14,82};
2814 y := 1;

2815 While(y <= Length(x))
2816 [
2817     Echo(y, "- ", x[y]);
2818     y := y + 1;
2819 ];

2820 %/mathpiper

2821 %output,preserve="false"
```

```
2822         Result: True
2823
2824     Side Effects:
2825     1 - 55
2826     2 - 93
2827     3 - 40
2828     4 - 21
2829     5 - 7
2830     6 - 24
2831     7 - 15
2832     8 - 14
2833     9 - 82
2834 .    %/output
```

2835 A **loop** can also be used to search through a list. The following program uses a
2836 **While()** function and an **If()** function to search through a list to see if it contains
2837 the number **53**. If 53 is found in the list, a message is printed:

```
2838 %mathpiper
2839 //Determine if 53 is in the list.
2840 testList := {18,26,32,42,53,43,54,6,97,41};
2841 index := 1;
2842 While(index <= Length(testList))
2843 [
2844     If(testList[index] = 53,
2845         Echo("53 was found in the list at position", index));
2846     index := index + 1;
2847 ];
2848
2849 %/mathpiper
2850 %output,preserve="false"
2851     Result: True
2852
2853     Side Effects:
2854     53 was found in the list at position 5
2855 .    %/output
```

2856 When this program was executed, it determined that **53** was present in the list at
2857 position **5**.

2858 15.2.1 Using A While Loop And Append() To Place Values In A List

2859 In an earlier section it was mentioned that it would be nice if we could store a set
2860 of values for later processing and this can be done with a **while loop** and the

2861 **Append()** function. The following program creates an empty list and assigned it
2862 to the variable **primes**. The **while loop** and the **IsPrime()** function is then used
2863 to locate the prime integers between 1 and 50 and the **Append()** function is used
2864 to place them in the list. The last part of the program then prints some
2865 information about the numbers that were placed into the list:

```
2866 %mathpiper
2867 //Place the prime numbers between 1 and 50 (inclusive) into a list.
2868 //Create an empty list.
2869 primes := {};
2870 x := 1;
2871 While(x <= 50)
2872 [
2873     /*
2874         If x is prime, append it to the end of the list and then assign
2875         the new list that is created to the variable 'primes'.
2876     */
2877     If(IsPrime(x), primes := Append(primes, x ) );
2878
2879     x := x + 1;
2880 ];
2881 //Print information about the primes that were found.
2882 Echo("Primes ", primes);
2883 Echo("The number of primes in the list = ", Length(primes) );
2884 Echo("The first number in the list = ", primes[1] );
2885 %/mathpiper
2886     %output,preserve="false"
2887     Result: True
2888
2889     Side Effects:
2890     Primes {2,3,5,7,11,13,17,19,23,29,31,37,41,43,47}
2891     The number of primes in the list = 15
2892     The first number in the list = 2
2893 . %/output
```

2894 The ability to place values into a list with a loop is very powerful and we will be
2895 using this ability throughout the rest of the book.

2896 15.3 Exercises

2897 For the following exercises, create a new MathRider worksheet file called
2898 **book_1_section_15a_exercises_<your first name>_<your last name>.mrw.**

2899 **(Note: there are no spaces in this file name)**. For example, John Smith's
2900 worksheet would be called:

2901 **book_1_section_15a_exercises_john_smith.mrw.**

2902 After this worksheet has been created, place your answer for each exercise that
2903 requires a fold into its own fold in this worksheet. Place a title attribute in the
2904 start tag of each fold which indicates the exercise the fold contains the solution
2905 to. The folds you create should look similar to this one:

2906 `%mathpiper,title="Exercise 1"`

2907 `//Sample fold.`

2908 `%/mathpiper`

2909 If an exercise uses the MathPiper console instead of a fold, copy the work you
2910 did in the console into the worksheet so it can be saved but do not put it in a fold.

2911 **15.3.1 Exercise 1**

2912 Carefully read all of section 15 up to this point. Evaluate each one of
2913 the examples in the sections you read in the MathPiper worksheet you
2914 created or in the MathPiper console and verify that the results match the
2915 ones in the book. Copy all of the console examples you evaluated into your
2916 worksheet so they will be saved but do not put them in a fold.

2917 **15.3.2 Exercise 2**

2918 Create a program that uses a loop and an IsOdd() function to analyze the
2919 following list and then print the number of odd numbers it contains.

2920 `{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}`

2921 **15.3.3 Exercise 3**

2922 Create a program that uses a loop and an IsNegativeNumber() function to
2923 copy all of the negative numbers in the following list into a new list.
2924 Use the variable **negativeNumbers** to hold the new list.

2925 `{36,-29,-33,-6,14,7,-16,-3,-14,37,-38,-8,-45,-21,-26,6,6,38,-20,33,41,-`
2926 `4,24,37,40,29}`

2927 **15.3.4 Exercise 4**

2928 Create a program that uses a loop to analyze the following list and then
2929 print the following information about it:

- 2930 1) The largest number in the list.
2931 2) The smallest number in the list.
2932 3) The sum of all the numbers in the list.

```
2933 {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}
```

2934 **15.4 The ForEach() Looping Function**

2935 The **ForEach()** function uses a **loop** to index through a list like the While()
2936 function does, but it is more flexible and automatic. ForEach() also uses bodied
2937 notation like the While() function and here is its calling format:

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

2938 **ForEach()** selects each expression in a list in turn, assigns it to the passed-in
2939 "variable", and then executes the expressions that are inside of "body".
2940 Therefore, body is **executed once for each expression in the list**.

2941 **15.5 Print All The Values In A List Using A ForEach() function**

2942 This example shows how ForEach() can be used to print all of the items in a list:

```
2943 %mathpiper
```

```
2944 //Print all values in a list.
```

```
2945 ForEach(value, {50,51,52,53,54,55,56,57,58,59})
```

```
2946 [  
2947     Echo(value);  
2948 ];
```

```
2949 %/mathpiper
```

```
2950     %output,preserve="false"
```

```
2951     Result: True
```

```
2952  
2953     Side Effects:
```

```
2954     50
```

```
2955     51
```

```
2956     52
```

```
2957     53
```

```
2958     54
```

```
2959     55
```

```
2960     56
```

```
2961     57
```

```
2962     58
```

```
2963     59
```

```
2964 .    %/output
```

2965 **15.6 Calculate The Sum Of The Numbers In A List Using ForEach()**

2966 In previous examples, counting code in the form **x := x + 1** was used to count
2967 how many times a while loop was executed. The following program uses a
2968 **ForEach()** function and a line of code similar to this counter to calculate the
2969 **sum of the numbers in a list**:

```
2970 %mathpiper
2971 /*
2972    This program calculates the sum of the numbers
2973    in a list.
2974 */
2975 //This variable is used to accumulate the sum.
2976 sum := 0;
2977 ForEach(x, {1,2,3,4,5,6,7,8,9,10} )
2978 [
2979     /*
2980        Add the contents of x to the contents of sum
2981        and place the result back into sum.
2982     */
2983     sum := sum + x;
2984
2985     //Print the sum as it is being accumulated.
2986     Write(sum,,);
2987 ];
2988 NewLine(); NewLine();
2989 Echo("The sum of the numbers in the list = ", sum);
2990 %/mathpiper
2991 %output,preserve="false"
2992 Result: True
2993
2994 Side Effects:
2995 1,3,6,10,15,21,28,36,45,55,
2996
2997 The sum of the numbers in the list = 55
2998 . %/output
```

2999 In the above program, the integers **1** through **10** were manually placed into a list
3000 by typing them individually. This method is limited because only a relatively
3001 small number of integers can be placed into a list this way. The following section
3002 discusses an operator which can be used to automatically place a large number
3003 of integers into a list with very little typing.

3004 **15.7 The .. Range Operator**

```
first .. last
```

3005 A programmer often needs to create a list which contains **consecutive integers**
3006 and the **.. "range"** operator can be used to do this. The **first** integer in the list is
3007 placed before the **..** operator and the **last** integer in the list is placed after it
3008 (**Note: there must be a space immediately to the left of the .. operator**
3009 **and a space immediately to the right of it or an error will be generated.**).
3010 Here are some examples:

```
3011 In> 1 .. 10
3012 Result> {1,2,3,4,5,6,7,8,9,10}

3013 In> 10 .. 1
3014 Result> {10,9,8,7,6,5,4,3,2,1}

3015 In> 1 .. 100
3016 Result> {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
3017          21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,
3018          38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,
3019          55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,
3020          72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,
3021          89,90,91,92,93,94,95,96,97,98,99,100}

3022 In> -10 .. 10
3023 Result> {-10,-9,-8,-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,8,9,10}
```

3024 As these examples show, the **..** operator can generate lists of integers in
3025 ascending order and descending order. It can also generate lists that are very
3026 large and ones that contain negative integers.

3027 Remember, though, if one or both of the spaces around the **..** are omitted, an
3028 error is generated:

```
3029 In> 1..3
3030 Result>
3031 Error parsing expression, near token .3.
```

3032 **15.8 Using ForEach() With The Range Operator To Print The Prime**
3033 **Numbers Between 1 And 100**

3034 The following program shows how to use a **ForEach()** function instead of a
3035 **While()** function to print the prime numbers between 1 and 100. Notice that
3036 loops that are implemented with **ForEach()** often require less typing than
3037 their **While()** based equivalents:


```
3038 %mathpiper
3039 /*
3040 This program prints the prime integers between 1 and 100 using
3041 a ForEach() function instead of a While() function. Notice that
3042 the ForEach() version requires less typing than the While()
3043 version.
3044 */
3045 ForEach(x, 1 .. 100)
3046 [
3047     If(IsPrime(x), Write(x,,) );
3048 ];
3049 %/mathpiper
3050 %output,preserve="false"
3051 Result: True
3052
3053 Side Effects:
3054 2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,
3055 73,79,83,89,97,
3056 . %/output
```

3057 15.8.1 Using ForEach() And The Range Operator To Place The Prime 3058 Numbers Between 1 And 50 Into A List

3059 A ForEach() function can also be used to place values in a list, just the the
3060 While() function can:

```
3061 %mathpiper
3062 /*
3063 Place the prime numbers between 1 and 50 into
3064 a list using a ForEach() function.
3065 */
3066 //Create a new list.
3067 primes := {};
3068 ForEach(number, 1 .. 50)
3069 [
3070     /*
3071     If number is prime, append it to the end of the list and
3072     then assign the new list that is created to the variable
3073     'primes'.
3074     */
3075     If(IsPrime(number), primes := Append(primes, number) );
3076 ];
```

```
3077 //Print information about the primes that were found.
3078 Echo("Primes ", primes);
3079 Echo("The number of primes in the list = ", Length(primes) );
3080 Echo("The first number in the list = ", primes[1] );
3081 %/mathpiper
3082     %output,preserve="false"
3083     Result: True
3084
3085     Side Effects:
3086     Primes {2,3,5,7,11,13,17,19,23,29,31,37,41,43,47}
3087     The number of primes in the list = 15
3088     The first number in the list = 2
3089 .    %/output
```

3090 As can be seen from the above examples, the **ForEach()** function and the **range**
3091 **operator** can do a significant amount of work with very little typing. You will
3092 discover in the next section that MathPiper has functions which are even more
3093 powerful than these two.

3094 15.8.2 Exercises

3095 For the following exercises, create a new MathRider worksheet file called
3096 **book_1_section_15b_exercises_<your first name>_<your last name>.mrw.**
3097 (**Note: there are no spaces in this file name**). For example, John Smith's
3098 worksheet would be called:

3099 **book_1_section_15b_exercises_john_smith.mrw.**

3100 After this worksheet has been created, place your answer for each exercise that
3101 requires a fold into its own fold in this worksheet. Place a title attribute in the
3102 start tag of each fold which indicates the exercise the fold contains the solution
3103 to. The folds you create should look similar to this one:

```
3104 %mathpiper,title="Exercise 1"
3105 //Sample fold.
3106 %/mathpiper
```

3107 If an exercise uses the MathPiper console instead of a fold, copy the work you
3108 did in the console into the worksheet so it can be saved but do not put it in a fold.

3109 15.8.3 Exercise 1

3110 Carefully read all of section 15 starting at the end of the previous
3111 exercises and up to this point. Evaluate each one of the examples in the

3112 sections you read in the MathPiper worksheet you created or in the
3113 MathPiper console and verify that the results match the ones in the book.
3114 Copy all of the console examples you evaluated into your worksheet so they
3115 will be saved but do not put them in a fold.

3116 15.8.4 Exercise 2

3117 Create a program that uses a **ForEach()** function and an **IsOdd()** function to
3118 analyze the following list and then print the number of odd numbers it
3119 contains.

3120 {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}

3121 15.8.5 Exercise 3

3122 Create a program that uses a **ForEach()** function and an **IsNegativeNumber()**
3123 function to copy all of the negative numbers in the following list into a
3124 new list. Use the variable **negativeNumbers** to hold the new list.

3125 {36,-29,-33,-6,14,7,-16,-3,-14,37,-38,-8,-45,-21,-26,6,6,38,-20,33,41,-
3126 4,24,37,40,29}

3127 15.8.6 Exercise 4

3128 Create a program that uses a **ForEach()** function to analyze the following
3129 list and then print the following information about it:

- 3130 1) The largest number in the list.
3131 2) The smallest number in the list.
3132 3) The sum of all the numbers in the list.

3133 {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}

3134 15.8.7 Exercise 5

3135 Create a program that uses a **while loop** to make a list that contains **1000**
3136 **random integers** between **1** and **100** inclusive. Then, use a **ForEach()**
3137 function to determine how many integers in the list are **prime** and use an
3138 **Echo()** function to print this total.

3139 **16 Functions & Operators Which Loop Internally**

3140 Looping is such a useful capability that MathPiper has many functions which
3141 loop internally. Now that you have some experience with loops, you can use this
3142 experience to help you imagine how these functions use loops to process the
3143 information that is passed to them.

3144 **16.1 Functions & Operators Which Loop Internally To Process Lists**

3145 This section discusses a number of functions that use loops to process lists.

3146 **16.1.1 TableForm()**

```
TableForm(list)
```

3147 The **TableForm()** function prints the contents of a list in the form of a table.
3148 Each member in the list is printed on its own line and this sometimes makes the
3149 contents of the list easier to read:

```
3150 In> testList := {2,4,6,8,10,12,14,16,18,20}  
3151 Result> {2,4,6,8,10,12,14,16,18,20}
```

```
3152 In> TableForm(testList)  
3153 Result> True  
3154 Side Effects>  
3155 2  
3156 4  
3157 6  
3158 8  
3159 10  
3160 12  
3161 14  
3162 16  
3163 18  
3164 20
```

3165 **16.1.2 Contains()**

3166 The **Contains()** function searches a list to determine if it contains a given
3167 expression. If it finds the expression, it returns **True** and if it doesn't find the
3168 expression, it returns **False**. Here is the calling format for Contains():

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

3169 The following code shows Contains() being used to locate a number in a list:

```
3170 In> Contains({50,51,52,53,54,55,56,57,58,59}, 53)
3171 Result> True
```

```
3172 In> Contains({50,51,52,53,54,55,56,57,58,59}, 75)
3173 Result> False
```

3174 The **Not()** function can also be used with predicate functions like Contains() to
3175 change their results to the opposite truth value:

```
3176 In> Not Contains({50,51,52,53,54,55,56,57,58,59}, 75)
3177 Result> True
```

3178 16.1.3 Find()

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

3179 The **Find()** function searches a list for the first occurrence of a given expression.
3180 If the expression is found, the **position of its first occurrence** is returned and
3181 if it is not found, **-1** is returned:

```
3182 In> Find({23, 15, 67, 98, 64}, 15)
3183 Result> 2
```

```
3184 In> Find({23, 15, 67, 98, 64}, 8)
3185 Result> -1
```

3186 16.1.4 Count()

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

3187 **Count()** determines the number of times a given expression occurs in a list:

```
3188 In> testList := {a,b,b,c,c,c,d,d,d,d,e,e,e,e,e}
3189 Result> {a,b,b,c,c,c,d,d,d,d,e,e,e,e,e}
```

```
3190 In> Count(testList, c)
3191 Result> 3
```

```
3192 In> Count(testList, e)
3193 Result> 5
```

```
3194 In> Count(testList, z)
3195 Result> 0
```

3196 **16.1.5 Select()**

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

3197 **Select()** returns a list that contains all the expressions in a list which make a
3198 given predicate function return **True**:

```
3199 In> Select("IsPositiveInteger", {46,87,59,-27,11,86,-21,-58,-86,-52})  
3200 Result> {46,87,59,11,86}
```

3201 In this example, notice that the **name** of the predicate function is passed to
3202 **Select()** in **double quotes**. There are other ways to pass a predicate function to
3203 **Select()** but these are covered in a later section.

3204 Here are some further examples which use the **Select()** function:

```
3205 In> Select("IsOdd", {16,14,82,92,33,74,99,67,65,52})  
3206 Result> {33,99,67,65}
```

```
3207 In> Select("IsEven", {16,14,82,92,33,74,99,67,65,52})  
3208 Result> {16,14,82,92,74,52}
```

```
3209 In> Select("IsPrime", 1 .. 75)  
3210 Result> {2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73}
```

3211 Notice how the third example uses the **..** operator to automatically generate a list
3212 of consecutive integers from 1 to 75 for the **Select()** function to analyze.

3213 **16.1.6 The Nth() Function & The [] Operator**

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

3214 The **Nth()** function simply returns the expression which is at a given position in
3215 a list. This example shows the **third** expression in a list being obtained:

```
3216 In> testList := {a,b,c,d,e,f,g}  
3217 Result> {a,b,c,d,e,f,g}
```

```
3218 In> Nth(testList, 3)  
3219 Result> c
```

3220 As discussed earlier, the **[]** operator can also be used to obtain a single
3221 expression from a list:

```
3222 In> testList[3]
3223 Result> c
```

3224 The **[]** operator can even obtain a single expression directly from a list without
3225 needing to use a variable:

```
3226 In> {a,b,c,d,e,f,g}[3]
3227 Result> c
```

3228 16.1.7 The : Prepend Operator

```
expression : list
```

3229 The prepend operator is a colon **:** and it can be used to add an expression to the
3230 beginning of a list:

```
3231 In> testList := {b,c,d}
3232 Result> {b,c,d}

3233 In> testList := a:testList
3234 Result> {a,b,c,d}
```

3235 16.1.8 Concat()

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

3236 The Concat() function is short for "concatenate" which means to join together
3237 sequentially. It takes two or more lists and joins them together into a single
3238 larger list:

```
3239 In> Concat({a,b,c}, {1,2,3}, {x,y,z})
3240 Result> {a,b,c,1,2,3,x,y,z}
```

3241 16.1.9 Insert(), Delete(), & Replace()

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

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

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

3242 **Insert()** inserts an expression into a list at a given index, **Delete()** deletes an
3243 expression from a list at a given index, and **Replace()** replaces an expression in
3244 a list at a given index with another expression:

```
3245 In> testList := {a,b,c,d,e,f,g}
3246 Result> {a,b,c,d,e,f,g}

3247 In> testList := Insert(testList, 4, 123)
3248 Result> {a,b,c,123,d,e,f,g}

3249 In> testList := Delete(testList, 4)
3250 Result> {a,b,c,d,e,f,g}

3251 In> testList := Replace(testList, 4, xxx)
3252 Result> {a,b,c,xxx,e,f,g}
```

3253 16.1.10 Take()

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

3254 **Take()** obtains a sublist from the **beginning** of a list, the **end** of a list, or the
3255 **middle** of a list. The expressions in the list that are not taken are discarded.

3256 A **positive** integer passed to Take() indicates how many expressions should be
3257 taken from the **beginning** of a list:

```
3258 In> testList := {a,b,c,d,e,f,g}
3259 Result> {a,b,c,d,e,f,g}

3260 In> Take(testList, 3)
3261 Result> {a,b,c}
```

3262 A **negative** integer passed to Take() indicates how many expressions should be
3263 taken from the **end** of a list:

```
3264 In> Take(testList, -3)
3265 Result> {e,f,g}
```

3266 Finally, if a **two member list** is passed to Take() it indicates the **range** of
3267 expressions that should be taken from the **middle** of a list. The **first** value in the
3268 passed-in list specifies the **beginning** index of the range and the **second** value
3269 specifies its **end**:

```
3270 In> Take(testList, {3,5})
3271 Result> {c,d,e}
```


3272 **16.1.11 Drop()**

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

3273 **Drop()** does the opposite of Take() in that it **drops** expressions from the
3274 **beginning** of a list, the **end** of a list, or the **middle** of a list and **returns a list**
3275 **which contains the remaining expressions.**

3276 A **positive** integer passed to Drop() indicates how many expressions should be
3277 dropped from the **beginning** of a list:

```
3278 In> testList := {a,b,c,d,e,f,g}
3279 Result> {a,b,c,d,e,f,g}
```

```
3280 In> Drop(testList, 3)
3281 Result> {d,e,f,g}
```

3282 A **negative** integer passed to Drop() indicates how many expressions should be
3283 dropped from the **end** of a list:

```
3284 In> Drop(testList, -3)
3285 Result> {a,b,c,d}
```

3286 Finally, if a **two member list** is passed to Drop() it indicates the **range** of
3287 expressions that should be dropped from the **middle** of a list. The **first** value in
3288 the passed-in list specifies the **beginning** index of the range and the **second**
3289 value specifies its **end**:

```
3290 In> Drop(testList, {3,5})
3291 Result> {a,b,f,g}
```

3292 **16.1.12 FillList()**

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

3293 The FillList() function simply creates a list which is of size "length" and fills it
3294 with "length" copies of the given expression:

```
3295 In> FillList(a, 5)
3296 Result> {a,a,a,a,a}
```

```
3297 In> FillList(42,8)
3298 Result> {42,42,42,42,42,42,42,42}
```

3299 **16.1.13 RemoveDuplicates()**

```
RemoveDuplicates(list)
```

3300 **RemoveDuplicates()** removes any duplicate expressions that are contained in a
3301 list:

```
3302 In> testList := {a,a,b,c,c,b,b,a,b,c,c}
```

```
3303 Result> {a,a,b,c,c,b,b,a,b,c,c}
```

```
3304 In> RemoveDuplicates(testList)
```

```
3305 Result> {a,b,c}
```

3306 **16.1.14 Reverse()**

```
Reverse(list)
```

3307 **Reverse()** reverses the order of the expressions in a list:

```
3308 In> testList := {a,b,c,d,e,f,g,h}
```

```
3309 Result> {a,b,c,d,e,f,g,h}
```

```
3310 In> Reverse(testList)
```

```
3311 Result> {h,g,f,e,d,c,b,a}
```

3312 **16.1.15 Partition()**

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

3313 The **Partition()** function breaks a list into sublists of size "partition_size":

```
3314 In> testList := {a,b,c,d,e,f,g,h}
```

```
3315 Result> {a,b,c,d,e,f,g,h}
```

```
3316 In> Partition(testList, 2)
```

```
3317 Result> {{a,b},{c,d},{e,f},{g,h}}
```

3318 If the partition_size does not divide the length of the list **evenly**, the remaining
3319 elements are discarded:

```
3320 In> Partition(testList, 3)
```

```
3321 Result> {{h,b,c},{d,e,f}}
```

3322 The number of elements that Partition() will discard can be calculated by
3323 dividing the length of a list by the partition size and obtaining the **remainder**:

```
3324 In> Length(testList) % 3  
3325 Result> 2
```

3326 Remember that % is the remainder operator. It divides two integers and returns
3327 their remainder.

3328 16.1.16 Table()

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

3329 The Table() function creates a list of values by doing the following:

- 3330 1) Generating a sequence of values between a "begin_value" and an
3331 "end_value" with each value being incremented by the "step_amount".
- 3332 2) Placing each value in the sequence into the specified "variable", one value
3333 at a time.
- 3334 3) Evaluating the defined "expression" (which contains the defined "variable")
3335 for each value, one at a time.
- 3336 4) Placing the result of each "expression" evaluation into the result list.

3337 This example generates a list which contains the integers 1 through 10:

```
3338 In> Table(x, x, 1, 10, 1)  
3339 Result> {1,2,3,4,5,6,7,8,9,10}
```

3340 Notice that the expression in this example is simply the variable 'x' itself with no
3341 other operations performed on it.

3342 The following example is similar to the previous one except that its expression
3343 multiplies 'x' by 2:

```
3344 In> Table(x*2, x, 1, 10, 1)  
3345 Result> {2,4,6,8,10,12,14,16,18,20}
```

3346 Lists which contain decimal values can also be created by setting the
3347 "step_amount" to a decimal:

```
3348 In> Table(x, x, 0, 1, .1)  
3349 Result> {0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1}
```

3350 **16.1.17 HeapSort()**

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

3351 **HeapSort()** sorts the elements of **list** into the order indicated by **compare** with
3352 **compare** typically being the **less than** operator "<" or the **greater than**
3353 operator ">":

```
3354 In> HeapSort({4,7,23,53,-2,1}, "<");  
3355 Result: {-2,1,4,7,23,53}
```

```
3356 In> HeapSort({4,7,23,53,-2,1}, ">");  
3357 Result: {53,23,7,4,1,-2}
```

```
3358 In> HeapSort({1/2,3/5,7/8,5/16,3/32}, "<")  
3359 Result: {3/32,5/16,1/2,3/5,7/8}
```

```
3360 In> HeapSort({.5,3/5,.76,5/16,3/32}, "<")  
3361 Result: {3/32,5/16,.5,3/5,.76}
```

3362 **16.2 Functions That Work With Integers**

3363 This section discusses various functions which work with integers. Some of
3364 these functions also work with non-integer values and their use with non-
3365 integers is discussed in other sections.

3366 **16.2.1 RandomIntegerVector()**

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

3367 A vector is a list that does not contain other lists. **RandomIntegerVector()**
3368 creates a list of size "length" that contains random integers that are no lower
3369 than "lowest_possible" and no higher than "highest possible". The following
3370 example creates **10** random integers between **1** and **99** inclusive:

```
3371 In> RandomIntegerVector(10, 1, 99)  
3372 Result> {73,93,80,37,55,93,40,21,7,24}
```

3373 **16.2.2 Max() & Min()**

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

3374 If two values are passed to **Max()**, it determines which one is larger:

```
3375 In> Max(10, 20)
```

3376 `Result> 20`

3377 If a list of values are passed to `Max()`, it finds the largest value in the list:

3378 `In> testList := RandomIntegerVector(10, 1, 99)`

3379 `Result> {73,93,80,37,55,93,40,21,7,24}`

3380 `In> Max(testList)`

3381 `Result> 93`

3382 The **Min()** function is the opposite of the `Max()` function.

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

```
Min(list)
```

3383 If two values are passed to `Min()`, it determines which one is smaller:

3384 `In> Min(10, 20)`

3385 `Result> 10`

3386 If a list of values are passed to `Min()`, it finds the smallest value in the list:

3387 `In> testList := RandomIntegerVector(10, 1, 99)`

3388 `Result> {73,93,80,37,55,93,40,21,7,24}`

3389 `In> Min(testList)`

3390 `Result> 7`

3391 **16.2.3 Div() & Mod()**

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

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

3392 **Div()** stands for "divide" and determines the whole number of times a divisor
3393 goes into a dividend:

3394 `In> Div(7, 3)`

3395 `Result> 2`

3396 **Mod()** stands for "modulo" and it determines the remainder that results when a
3397 dividend is divided by a divisor:

3398 `In> Mod(7,3)`

3399 `Result> 1`

3400 The remainder/modulo operator **%** can also be used to calculate a remainder:

```
3401 In> 7 % 2
3402 Result> 1
```

3403 **16.2.4 Gcd()**

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

3404 GCD stands for Greatest Common Divisor and the **Gcd()** function determines the
3405 greatest common divisor of the values that are passed to it.

3406 If two integers are passed to Gcd(), it calculates their greatest common divisor:

```
3407 In> Gcd(21, 56)
3408 Result> 7
```

3409 If a list of integers are passed to Gcd(), it finds the greatest common divisor of all
3410 the integers in the list:

```
3411 In> Gcd({9, 66, 123})
3412 Result> 3
```

3413 **16.2.5 Lcm()**

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

3414 LCM stands for Least Common Multiple and the **Lcm()** function determines the
3415 least common multiple of the values that are passed to it.

3416 If two integers are passed to Lcm(), it calculates their least common multiple:

```
3417 In> Lcm(14, 8)
3418 Result> 56
```

3419 If a list of integers are passed to Lcm(), it finds the least common multiple of all
3420 the integers in the list:

```
3421 In> Lcm({3, 7, 9, 11})
3422 Result> 693
```

3423 **16.2.6 Sum()**

```
Sum(list)
```

3424 **Sum()** can find the sum of a list that is passed to it:

3425 In> testList := RandomIntegerVector(10,1,99)

3426 Result> {73,93,80,37,55,93,40,21,7,24}

3427 In> Sum(testList)

3428 Result> 523

3429 In> testList := 1 .. 10

3430 Result> {1,2,3,4,5,6,7,8,9,10}

3431 In> Sum(testList)

3432 Result> 55

3433 **16.2.7 Product()**

```
Product(list)
```

3434 This function has two calling formats, only one of which is discussed here.

3435 **Product(list)** multiplies all the expressions in a list together and returns their
3436 product:

3437 In> Product({1,2,3})

3438 Result> 6

3439 **16.3 Exercises**

3440 For the following exercises, create a new MathRider worksheet file called
3441 **book_1_section_16_exercises_<your first name>_<your last name>.mrw.**
3442 (**Note: there are no spaces in this file name**). For example, John Smith's
3443 worksheet would be called:

3444 **book_1_section_16_exercises_john_smith.mrw.**

3445 After this worksheet has been created, place your answer for each exercise that
3446 requires a fold into its own fold in this worksheet. Place a title attribute in the
3447 start tag of each fold which indicates the exercise the fold contains the solution
3448 to. The folds you create should look similar to this one:

3449 %mathpiper,title="Exercise 1"

3450 //Sample fold.

3451 `%/mathpiper`

3452 If an exercise uses the MathPiper console instead of a fold, copy the work you
3453 did in the console into the worksheet so it can be saved but do not put it in a fold.

3454 **16.3.1 Exercise 1**

3455 Carefully read all of section 16 up to this point. Evaluate each one of
3456 the examples in the sections you read in the MathPiper worksheet you
3457 created or in the MathPiper console and verify that the results match the
3458 ones in the book. Copy all of the console examples you evaluated into your
3459 worksheet so they will be saved but do not put them in a fold.

3460 **16.3.2 Exercise 2**

3461 Create a program that uses `RandomIntegerVector()` to create a 100 member
3462 list that contains random integers between 1 and 5 inclusive. Use `Count()`
3463 to determine how many of each digit 1-5 are in the list and then print this
3464 information. Hint: you can use the `HeapSort()` function to sort the
3465 generated list to make it easier to check if your program is counting
3466 correctly.

3467 **16.3.3 Exercise 3**

3468 Create a program that uses `RandomIntegerVector()` to create a 100 member
3469 list that contains random integers between 1 and 50 inclusive and use
3470 `Contains()` to determine if the number 25 is in the list. Print "25 was in
3471 the list." if 25 was found in the list and "25 was not in the list." if it
3472 wasn't found.

3473 **16.3.4 Exercise 4**

3474 Create a program that uses `RandomIntegerVector()` to create a 100 member
3475 list that contains random integers between 1 and 50 inclusive and use
3476 `Find()` to determine if the number 10 is in the list. Print the position of
3477 10 if it was found in the list and "10 was not in the list." if it wasn't
3478 found.

3479 **16.3.5 Exercise 5**

3480 Create a program that uses `RandomIntegerVector()` to create a 100 member
3481 list that contains random integers between 0 and 3 inclusive. Use `Select()`
3482 with the `IsNonZeroInteger()` predicate function to obtain all of the nonzero
3483 integers in this list.

3484 **16.3.6 Exercise 6**

3485 Create a program that uses `Table()` to obtain a list which contains the
3486 squares of the integers between 1 and 10 inclusive.

3487 17 Nested Loops

3488 Now that you have seen how to solve problems with single loops, it is time to
3489 discuss what can be done when a loop is placed inside of another loop. A loop
3490 that is placed **inside** of another loop it is called a **nested loop** and this nesting
3491 can be extended to numerous levels if needed. This means that loop 1 can have
3492 loop 2 placed inside of it, loop 2 can have loop 3 placed inside of it, loop 3 can
3493 have loop 4 placed inside of it, and so on.

3494 Nesting loops allows the programmer to accomplish an enormous amount of
3495 work with very little typing.

3496 17.1 Generate All The Combinations That Can Be Entered Into A Two Digit 3497 Wheel Lock Using A Nested Loop



3498 The following program generates all the combinations that can be entered into a
3499 two digit wheel lock. It uses a nested loop to accomplish this with the "**inside**"
3500 nested loop being used to generate **one's place** digits and the "**outside**" loop
3501 being used to generate **ten's place** digits.

```
3502 %mathpiper
3503 /*
3504    Generate all the combinations can be entered into a two
3505    digit wheel lock.
3506 */
3507 combinations := {};
3508 ForEach(digit1, 0 .. 9) //This loop is called the "outside" loop.
```

```
3509 [
3510     ForEach(digit2, 0 .. 9)//This loop is called the "inside" loop.
3511     [
3512         combinations := Append(combinations, {digit1, digit2});
3513     ];
3514 ];

3515 Echo(TableForm(combinations));

3516 %/mathpiper

3517     %output,preserve="false"
3518     Result: True
3519
3520     Side Effects:
3521     {0,0}
3522     {0,1}
3523     {0,2}
3524     {0,3}
3525     {0,4}
3526     {0,5}
3527     {0,6}
3528     .
3529     . //The middle of the list has not been shown.
3530     .
3531     {9,3}
3532     {9,4}
3533     {9,5}
3534     {9,6}
3535     {9,7}
3536     {9,8}
3537     {9,9}
3538     True
3539     %/output
```

3540 The relationship between the outside loop and the inside loop is interesting
3541 because each time the **outside loop cycles once**, the **inside loop cycles 10**
3542 **times**. Study this program carefully because nested loops can be used to solve a
3543 wide range of problems and therefore understanding how they work is
3544 important.

3545 17.2 Exercises

3546 For the following exercises, create a new MathRider worksheet file called
3547 **book_1_section_17_exercises_<your first name>_<your last name>.mrw**.
3548 (**Note: there are no spaces in this file name**). For example, John Smith's
3549 worksheet would be called:

3550 **book_1_section_17_exercises_john_smith.mrw**.

3551 After this worksheet has been created, place your answer for each exercise that
3552 requires a fold into its own fold in this worksheet. Place a title attribute in the
3553 start tag of each fold which indicates the exercise the fold contains the solution
3554 to. The folds you create should look similar to this one:

```
3555 %mathpiper,title="Exercise 1"
```

```
3556 //Sample fold.
```

```
3557 %/mathpiper
```

3558 If an exercise uses the MathPiper console instead of a fold, copy the work you
3559 did in the console into the worksheet so it can be saved but do not put it in a fold.

3560 **17.2.1 Exercise 1**

3561 Carefully read all of section 17 up to this point. Evaluate each one of
3562 the examples in the sections you read in the MathPiper worksheet you
3563 created or in the MathPiper console and verify that the results match the
3564 ones in the book. Copy all of the console examples you evaluated into your
3565 worksheet so they will be saved but do not put them in a fold.

3566 **17.2.2 Exercise 2**

3567 Create a program that will generate all of the combinations that can be
3568 entered into a three digit wheel lock. (Hint: a triple nested loop can be
3569 used to accomplish this.)

3570 18 User Defined Functions

3571 In computer programming, a **function** is a named section of code that can be
3572 **called** from other sections of code. **Values** can be sent to a function for
3573 processing as part of the **call** and a function always returns a value as its result.
3574 A function can also generate side effects when it is called and side effects have
3575 been covered in earlier sections.

3576 The values that are sent to a function when it is called are called **arguments** or
3577 **actual parameters** and a function can accept 0 or more of them. These
3578 arguments are placed within parentheses.

3579 MathPiper has many predefined functions (some of which have been discussed in
3580 previous sections) but users can create their own functions too. The following
3581 program creates a function called **addNums()** which takes two numbers as
3582 arguments, adds them together, and returns their sum back to the calling code
3583 as a result:

```
3584 In> addNums(num1,num2) := num1 + num2
3585 Result> True
```

3586 This line of code defined a new function called **addNums** and specified that it
3587 will accept two values when it is called. The **first** value will be placed into the
3588 variable **num1** and the **second** value will be placed into the variable **num2**.

3589 Variables like num1 and num2 which are used in a function to accept values from
3590 calling code are called **formal parameters**. **Formal parameter variables** are
3591 used inside a function to process the **values/actual parameters/arguments**
3592 that were placed into them by the calling code.

3593 The code on the **right side** of the **assignment operator** is **bound** to the
3594 function name "**addNums**" and it is executed each time **addNums()** is called.
3595 The following example shows the new **addNums()** function being called multiple
3596 times with different values being passed to it:

```
3597 In> addNums(2,3)
3598 Result> 5
```

```
3599 In> addNums(4,5)
3600 Result> 9
```

```
3601 In> addNums(9,1)
3602 Result> 10
```

3603 Notice that, unlike the functions that come with MathPiper, we chose to have this
3604 function's name start with a **lower case letter**. We could have had addNums()
3605 begin with an upper case letter but it is a **convention** in MathPiper for **user**

3606 **defined function names to begin with a lower case letter to distinguish**
3607 **them from the functions that come with MathPiper.**

3608 The values that are returned from user defined functions can also be assigned to
3609 variables. The following example uses a %mathpiper fold to define a function
3610 called **evenIntegers()** and then this function is used in the MathPiper console:

```
3611 %mathpiper
3612 evenIntegers(endInteger) :=
3613 [
3614     resultList := {};
3615     x := 2;
3616     While(x <= endInteger)
3617     [
3618         resultList := Append(resultList, x);
3619         x := x + 2;
3620     ];
3621 ];
3622 /*
3623 The result of the last expression which is executed in a function
3624 is the result that the function returns to the caller. In this case,
3625 resultList is purposely being executed last so that its contents are
3626 returned to the caller.
3627 */
3628 resultList;
3629 ];
3630 ];
3631 %/mathpiper
3632 %output,preserve="false"
3633 Result: True
3634 . %/output
3635 In> a := evenIntegers(10)
3636 Result> {2,4,6,8,10}
3637 In> Length(a)
3638 Result> 5
```

3639 The function **evenIntegers()** returns a list which contains all the even integers
3640 from 2 up through the value that was passed into it. The fold was first executed
3641 in order to define the **evenIntegers()** function and make it ready for use. The
3642 **evenIntegers()** function was then called from the MathPiper console and 10
3643 was passed to it.

3644 After the function was finished executing, it returned a list of even integers as a

3645 result and this result was assigned to the variable 'a'. We then passed the list
3646 that was assigned to 'a' to the **Length()** function in order to determine its size.

3647 **18.1 Global Variables, Local Variables, & Local()**

3648 The new **evenIntegers()** function seems to work well, but there is a problem.
3649 The variables 'x' and **resultList** were defined inside the function as **global**
3650 **variables** which means they are accessible from anywhere, including from
3651 within other functions, within other folds (as shown here):

```
3652 %mathpiper
3653 Echo(x, ",", resultList);
3654 %/mathpiper
3655     %output,preserve="false"
3656     Result: True
3657
3658     Side Effects:
3659     12 , {2,4,6,8,10}
3660 .    %/output
```

3661 and from within the MathPiper console:

```
3662 In> x
3663 Result> 12
3664 In> resultList
3665 Result> {2,4,6,8,10}
```

3666 **Using global variables inside of functions is usually not a good idea**
3667 because code in other functions and folds might already be using (or will use) the
3668 same variable names. Global variables which have the same name are the same
3669 variable. When one section of code changes the value of a given global variable,
3670 the value is changed everywhere that variable is used and this will eventually
3671 cause problems.

3672 In order to prevent errors being caused by global variables having the same
3673 name, a function named **Local()** can be called inside of a function to define what
3674 are called **local variables**. A **local variable** is only accessible inside the
3675 function it has been defined in, even if it has the same name as a global variable.
3676 The following example shows a second version of the **evenIntegers()** function
3677 which uses **Local()** to make 'x' and **resultList** local variables:

```
3678 %mathpiper
3679 /*
3680  This version of evenIntegers() uses Local() to make
3681  x and resultList local variables
3682  */
3683 evenIntegers(endInteger) :=
3684 [
3685     Local(x,resultList);
3686     resultList := {};
3687
3688     x := 2;
3689
3690     While(x <= endInteger)
3691     [
3692         resultList := Append(resultList, x);
3693         x := x + 2;
3694     ];
3695
3696     /*
3697     The result of the last expression which is executed in a function
3698     is the result that the function returns to the caller.  In this case,
3699     resultList is purposely being executed last so that its contents are
3700     returned to the caller.
3701     */
3702     resultList;
3703 ];
3704 %/mathpiper
3705     %output,preserve="false"
3706     Result: True
3707 .    %/output
```

3708 We can verify that '**x**' and **resultList** are now local variables by first clearing
3709 them, calling **evenIntegers()**, and then seeing what '**x**' and **resultList** contain:

```
3710 In> Clear(x, resultList)
3711 Result> True
3712 In> evenIntegers(10)
3713 Result> {2,4,6,8,10}
3714 In> x
3715 Result> x
3716 In> resultList
3717 Result> resultList
```

3718 18.2 Exercises

3719 For the following exercises, create a new MathRider worksheet file called
3720 **book_1_section_18_exercises_<your first name>_<your last name>.mrw.**
3721 **(Note: there are no spaces in this file name).** For example, John Smith's
3722 worksheet would be called:

3723 **book_1_section_18_exercises_john_smith.mrw.**

3724 After this worksheet has been created, place your answer for each exercise that
3725 requires a fold into its own fold in this worksheet. Place a title attribute in the
3726 start tag of each fold which indicates the exercise the fold contains the solution
3727 to. The folds you create should look similar to this one:

3728 `%mathpiper,title="Exercise 1"`

3729 `//Sample fold.`

3730 `%/mathpiper`

3731 If an exercise uses the MathPiper console instead of a fold, copy the work you
3732 did in the console into the worksheet so it can be saved but do not put it in a fold.

3733 18.2.1 Exercise 1

3734 Carefully read all of section 18 up to this point. Evaluate each one of
3735 the examples in the sections you read in the MathPiper worksheet you
3736 created or in the MathPiper console and verify that the results match the
3737 ones in the book. Copy all of the console examples you evaluated into your
3738 worksheet so they will be saved but do not put them in a fold.

3739 18.2.2 Exercise 2

3740 Create a function called **tenOddIntegers()** which returns a list which
3741 contains 10 random odd integers between 1 and 99 inclusive.

3742 18.2.3 Exercise 3

3743 Create a function called **convertStringToList(string)** which takes a string
3744 as a parameter and returns a list which contains all of the characters in
3745 the string. Here is an example of how the function should work:

3746 `In> convertStringToList("Hello friend!")`
3747 `Result> {"H","e","l","l","o"," ","f","r","i","e","n","d","!"}`

3748 `In> convertStringToList("Computer Algebra System")`
3749 `Result> {"C","o","m","p","u","t","e","r"," ","A","l","g","e","b","r","a","",`
3750 `","S","y","s","t","e","m"}`

3751 19 Miscellaneous topics

3752 19.1 Incrementing And Decrementing Variables With The ++ And -- 3753 Operators

3754 Up until this point we have been adding 1 to a variable with code in the form of **x**
3755 **:= x + 1** and subtracting 1 from a variable with code in the form of **x := x - 1**.
3756 Another name for **adding** 1 to a variable is **incrementing** it and **decrementing**
3757 a variable means to **subtract** 1 from it. Now that you have had some experience
3758 with these longer forms, it is time to show you shorter versions of them.

3759 19.1.1 Incrementing Variables With The ++ Operator

3760 The number 1 can be added to a variable by simply placing the ++ operator after
3761 it like this:

```
3762 In> x := 1  
3763 Result: 1
```

```
3764 In> x++;  
3765 Result: True
```

```
3766 In> x  
3767 Result: 2
```

3768 Here is a program that uses the ++ operator to increment a loop index variable:

```
3769 %mathpiper  
3770 count := 1;  
3771 While(count <= 10)  
3772 [  
3773     Echo(count);  
3774     count++; //The ++ operator increments the count variable.  
3775 ];  
3776  
3777 %/mathpiper  
3778 %output,preserve="false"  
3779 Result: True  
3780  
3781 Side Effects:  
3782 1  
3783 2
```

```
3784      3
3785      4
3786      5
3787      6
3788      7
3789      8
3790      9
3791     10
3792 .    %/output
```

3793 19.1.2 Decrementing Variables With The -- Operator

3794 The number 1 can be subtracted from a variable by simply placing the --
3795 operator after it like this:

```
3796 In> x := 1
3797 Result: 1

3798 In> x--;
3799 Result: True
```

```
3800 In> x
3801 Result: 0
```

3802 Here is a program that uses the -- operator to decrement a loop index variable:

```
3803 %mathpiper
3804 count := 10;
3805 While(count >= 1)
3806 [
3807     Echo(count);
3808
3809     count--; //The -- operator decrements the count variable.
3810 ];
```

```
3811 %/mathpiper

3812 %output,preserve="false"
3813 Result: True
3814
3815 Side Effects:
3816 10
3817 9
3818 8
3819 7
3820 6
3821 5
```

```
3822     4
3823     3
3824     2
3825     1
3826 .    %/output
```

3827 19.2 Exercises

3828 For the following exercises, create a new MathRider worksheet file called
3829 **book_1_section_19_exercises_<your first name>_<your last name>.mrw.**
3830 **(Note: there are no spaces in this file name).** For example, John Smith's
3831 worksheet would be called:

3832 **book_1_section_19_exercises_john_smith.mrw.**

3833 After this worksheet has been created, place your answer for each exercise that
3834 requires a fold into its own fold in this worksheet. Place a title attribute in the
3835 start tag of each fold which indicates the exercise the fold contains the solution
3836 to. The folds you create should look similar to this one:

```
3837 %mathpiper,title="Exercise 1"
```

```
3838 //Sample fold.
```

```
3839 %/mathpiper
```

3840 If an exercise uses the MathPiper console instead of a fold, copy the work you
3841 did in the console into the worksheet so it can be saved but do not put it in a fold.

3842 19.2.1 Exercise 1

3843 Carefully read all of section 19 up to this point. Evaluate each one of
3844 the examples in the sections you read in the MathPiper worksheet you
3845 created or in the MathPiper console and verify that the results match the
3846 ones in the book. Copy all of the console examples you evaluated into your
3847 worksheet so they will be saved but do not put them in a fold.