

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
3.1.1.1	The 64 Bit Version Of Windows Needs The 64 Bit Version Of Java	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.....	16
3.3	MathRider's Directory Structure & Execution Instructions.....	16
3.3.1	Executing MathRider On Windows Systems.....	17
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
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 Into A List.....	100
15.3 Exercises.....	101
15.3.1 Exercise 1.....	101
15.3.2 Exercise 2.....	101
15.3.3 Exercise 3.....	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.....	105
15.8.1 Using ForEach() And The Range Operator To Place The Prime Numbers Between 1 And 50 Into A List.....	106
15.8.2 Exercises.....	107
15.8.3 Exercise 1.....	107
15.8.4 Exercise 2.....	107
15.8.5 Exercise 3.....	107
15.8.6 Exercise 4.....	108
16 Functions & Operators Which Loop Internally.....	109
16.1 Functions & Operators Which Loop Internally To Process Lists.....	109
16.1.1 TableForm().....	109
16.1.2 Contains().....	109
16.1.3 Find().....	110
16.1.4 Count().....	110
16.1.5 Select().....	111
16.1.6 The Nth() Function & The [] Operator.....	111
16.1.7 The : Prepend Operator.....	112
16.1.8 Concat().....	112
16.1.9 Insert(), Delete(), & Replace().....	112

16.1.10 Take()	113
16.1.11 Drop()	114
16.1.12 FillList()	114
16.1.13 RemoveDuplicates()	115
16.1.14 Reverse()	115
16.1.15 Partition()	115
16.1.16 Table()	116
16.1.17 HeapSort()	117
16.2 Functions That Work With Integers	117
16.2.1 RandomIntegerVector()	117
16.2.2 Max() & Min()	117
16.2.3 Div() & Mod()	118
16.2.4 Gcd()	119
16.2.5 Lcm()	119
16.2.6 Sum()	120
16.2.7 Product()	120
16.3 Exercises	120
16.3.1 Exercise 1	121
16.3.2 Exercise 2	121
16.3.3 Exercise 3	121
16.3.4 Exercise 4	121
16.3.5 Exercise 5	121
17 Nested Loops	122
17.1 Generate All The Combinations That Can Be Entered Into A Two Digit Wheel Lock Using A Nested Loop	122
17.2 Exercises	123
17.2.1 Exercise 1	124
18 User Defined Functions	125
18.1 Global Variables, Local Variables, & Local()	127
18.2 Exercises	129
18.2.1 Exercise 1	129
18.2.2 Exercise 2	129
19 Miscellaneous topics	130
19.1 Incrementing And Decrementing Variables With The ++ And -- Operators	130
19.1.1 Incrementing Variables With The ++ Operator	130
19.1.2 Decrementing Variables With The -- Operator	131
19.1.3 The Break() Function	132
19.1.4 The Continue() Function	133
19.1.5 The Repeat() Function	133
19.1.6 The EchoTime() Function	135

19.2 Exercises.....	137
19.2.1 Exercise 1.....	137
19.2.2 Exercise 2.....	137
19.2.3 Exercise 3.....	138
19.2.4 Exercise 4.....	138
19.2.5 Exercise 5.....	138

1 Preface

2 1.1 Dedication

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

5 1.2 Acknowledgments

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

8 Susan Addington

9 Matthew Moelter

10 Sherm Ostrowsky

11 1.3 Support Email List

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

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

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

22 **2 Introduction**

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

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

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

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

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

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

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

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

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

57 and here is the same formula in text form:

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

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

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

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

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

80 **2.2 What Is MathRider?**

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

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

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

Table 1: MathRider user experience levels.

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

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

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

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

101 http://weblogs.java.net/blog/turbogeek/archive/2004/09/no_child_held_b_1.html

102 and Steve Yegge's thoughts on learning mathematics:

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

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

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

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

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

115 **3 Downloading And Installing MathRider**

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

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

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

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

124 <http://java.com/>

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

127 **3.1.1.1 *The 64 Bit Version Of Windows Needs The 64 Bit Version Of Java***

128 If you are using the 64 bit version of Windows, then you will need the 64 bit
129 version of Java. The 64 bit version of Java can be obtained here:

130 [https://cds.sun.com/is-bin/INTERSHOP.enfinity/WFS/CDS-CDS_Developer-
131 Site/en_US/-/USD/ViewProductDetail-Start?ProductRef=jre-6u16-oth-JPR@CDS-
132 CDS_Developer](https://cds.sun.com/is-bin/INTERSHOP.enfinity/WFS/CDS-CDS_Developer-Site/en_US/-/USD/ViewProductDetail-Start?ProductRef=jre-6u16-oth-JPR@CDS-CDS_Developer)

133 **3.1.2 Installing Java On A Macintosh**

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

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

138 **3.1.3 Installing Java On A Linux PC**

139 Locate the Java documentation for your Linux distribution and carefully follow
140 the instructions provided for installing a Java 6 compatible version of Java on
141 your system.

142 **3.2 *Downloading And Extracting***

143 One of the many benefits of learning MathRider is the programming-related

144 knowledge one gains about how open source software is developed on the
145 Internet. An important enabler of open source software development are
146 websites, such as sourceforge.net (<http://sourceforge.net>) and java.net
147 (<http://java.net>) which make software development tools available for free to
148 open source developers.

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

150 <http://mathrider.org>

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

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

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

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

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

173 3.2.1 Extracting The Archive File For Windows Users

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

178 After the extraction process is complete, a new folder called **mathrider** should
179 be present in the same folder that contains the archive file. **(Note: be careful**
180 **not to double click on the archive file by mistake when you are trying to**
181 **open the mathrider folder. The Windows operating system will open the**
182 **archive just like it opens folders and this can fool you into thinking you**

183 **are opening the mathrider folder when you are not. You may want to**
184 **move the archive file to another place on your hard drive after it has**
185 **been extracted to avoid this potential confusion.)**

186 3.2.2 Extracting The Archive File For Unix Users

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

190 `tar -xvjf <name of archive file>`

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

193 3.3 MathRider's Directory Structure & Execution Instructions

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

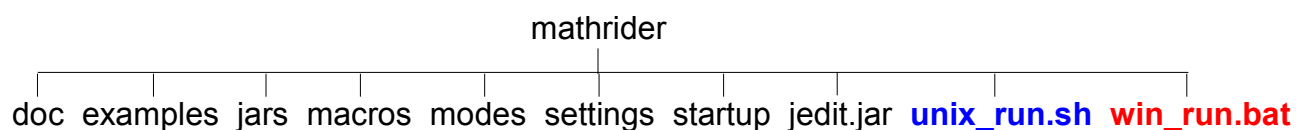


Illustration 1: MathRider's Directory Structure

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

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

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

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

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

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

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

204 **startup** - Contains startup scripts that are executed each time MathRider
205 launches.

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

207 **unix_run.sh** - The script used to execute MathRider on Unix systems.

208 **win_run.bat** - The batch file used to execute MathRider on Windows systems.

209 **3.3.1 Executing MathRider On Windows Systems**

210 Open the **mathrider** folder **(not the archive file!)** and double click on the
211 **win_run** file.

212 **3.3.2 Executing MathRider On Unix Systems**

213 Open a shell, change to the **mathrider** folder, and execute the **unix_run.sh**
214 script by typing the following:

215 `sh unix_run.sh`

216 **3.3.2.1 MacOS X**

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

220 `cd /Applications/mathrider`

221 Run mathrider by typing:

222 `sh unix_run.sh`

223 4 The Graphical User Interface

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

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

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

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

243 4.1 Buffers And Text Areas

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

250 4.2 The Gutter

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

254 4.3 Menus

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

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

261 4.3.1 File

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

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

267 4.3.2 Edit

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

273 4.3.3 Search

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

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

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

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

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

296 contents of the **Search for** text area and replace them with the contents of the
297 **Replace with** text area.

298 **4.3.4 Markers, Folding, and View**

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

300 **4.3.5 Utilities**

301 The utilities menu contains a significant number of actions, some that are useful
302 to beginners and others that are meant for experts. The two actions that are
303 most useful to beginners are the **Buffer Options** actions and the **Global**
304 **Options** actions. The **Buffer Options** actions allows the currently selected
305 buffer to be customized and the **Global Options** actions brings up a rich dialog
306 window that allows numerous aspects of the MathRider application to be
307 configured.

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

309 **4.3.6 Macros**

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

311 **4.3.7 Plugins**

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

319 **4.3.8 Help**

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

323 **4.4 The Toolbar**

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

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

331 **4.4.1 Undo And Redo**

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

337 **5 MathPiper: A Computer Algebra System For Beginners**

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

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

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

355 **5.1 Numeric Vs. Symbolic Computations**

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

360 **Symbolic** computations (which also called algebraic computations) relate "...to
361 the use of machines, such as computers, to manipulate mathematical equations
362 and expressions in symbolic form, as opposed to manipulating the
363 approximations of specific numerical quantities represented by those symbols."
364 (http://en.wikipedia.org/wiki/Symbolic_mathematics).

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

373 **5.2 Using The MathPiper Console As A Numeric (Scientific) Calculator**

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

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

```
381 MathPiper version ".76x".
```

```
382 In>
```

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

```
385 In> 2+2
```

```
386 Result> 4
```

```
387 In>
```

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

394 In addition to addition, MathPiper can also do subtraction, multiplication,
395 exponents, and division:

```
396 In> 5-2
```

```
397 Result> 3
```

```
398 In> 3*4
```

```
399 Result> 12
```

```
400 In> 2^3
```

```
401 Result> 8
```

```
402 In> 12/6
```

```
403 Result> 2
```

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

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

409 MathPiper can also work with decimal numbers:

```
410 In> .5+1.2  
411 Result> 1.7
```

```
412 In> 3.7-2.6  
413 Result> 1.1
```

```
414 In> 2.2*3.9  
415 Result> 8.58
```

```
416 In> 2.2^3  
417 Result> 10.648
```

```
418 In> 9.5/3.2  
419 Result> 9.5/3.2
```

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

```
423 In> N(9.5/3.2)  
424 Result> 2.96875
```

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

427 5.2.1 Functions

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

437 5.2.1.1 The Sqrt() Square Root Function

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


```
440 In> Sqrt(9)
441 Result: 3
```

```
442 In> Sqrt(8)
443 Result: Sqrt(8)
```

```
444 In> N(Sqrt(8))
445 Result: 2.828427125
```

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

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

```
454 2.828427125
```

```
455 2.82842712474619
```

```
456 2.82842712474619009760337744842
```

```
457 2.8284271247461900976033774484193961571393437507539
```

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

462 **5.2.1.2 The IsEven() Function**

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

```
465 In> IsEven(4)
466 Result> True
```

```
467 In> IsEven(5)
468 Result> False
```

469 MathPiper has a large number of functions some of which are described in more
470 depth in the MathPiper Documentation section and the MathPiper Programming
471 Fundamentals section. **A complete list of MathPiper's functions is**
472 **contained in the MathPiperDocs plugin and more of these functions will**
473 **be discussed soon.**

474 5.2.2 Accessing Previous Input And Results

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

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

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

```
485 In> 5*8  
486 Result> 40
```

```
487 In> %  
488 Result> 40
```

```
489 In> %*2  
490 Result> 80
```

491 5.3 Saving And Restoring A Console Session

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

497 5.3.1 Syntax Errors

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

```
503 In> 12 \ 6  
  
504 Error parsing expression, near token \
```

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

508 This section provided a short introduction to using MathPiper as a numeric
509 calculator and the next section contains a short introduction to using MathPiper
510 as a symbolic calculator.

511 **5.4 Using The MathPiper Console As A Symbolic Calculator**

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

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

```
517 In> 1/2 + 1/3  
518 Result> 5/6
```

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

```
523 In> %  
524 Result> 5/6
```

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

```
527 In> Numerator(%)  
528 Result> 5
```

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

532 **5.4.1 Variables**

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

539 'totalAmount', and 'loop6'.

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

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

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

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

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

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

554 In this example, the assignment operator (:=) was used to assign the result (or
555 **value**) $\frac{5}{6}$ to the variable 'a'. **When 'a' was evaluated by itself, the value it**
556 **was bound to (in this case $\frac{5}{6}$) was returned.** This value will stay bound to
557 the variable 'a' as long as MathPiper is running unless 'a' is cleared with the
558 **Clear()** function or 'a' has another value assigned to it. This is why we were able
559 to determine both the numerator and the denominator of the rational number
560 assigned to 'a' using two functions in turn. **(Note: there can be no spaces**
561 **between the : and the =)**

562 5.4.1.1 Calculating With Unbound Variables

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

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

```
566 In> a
567 Result> 9
```

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

```
570 In> Clear(a)
```

```
571 Result> True
```

```
572 In> a
```

```
573 Result> a
```

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

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

```
583 In> Factor(8)
```

```
584 Result> 2^3
```

```
585 In> Factor(14)
```

```
586 Result> 2*7
```

```
587 In> Factor(2343)
```

```
588 Result> 3*11*71
```

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

```
590 In> x
```

```
591 Result> x
```

```
592 In> IsBound(x)
```

```
593 Result> False
```

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

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

```
596 In> Expand(%)
```

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

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

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

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

610 **5.4.1.2 Variable And Function Names Are Case Sensitive**

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

```
617 In> Box := 1
618 Result> 1
```

```
619 In> box := 2
620 Result> 2
```

```
621 In> Box
622 Result> 1
```

```
623 In> box
624 Result> 2
```

625 **5.4.1.3 Using More Than One Variable**

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

```
628 a := 2
629 Result> 2
```

```
630 b := 3
631 Result> 3
```

```
632 a + b
633 Result> 5
```

```
634 answer := a + b
635 Result> 5
```

```
636 answer
637 Result> 5
```

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

640 the **left side** of the operator.

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

646 **5.5 Exercises**

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

649 **5.5.1 Exercise 1**

650 Carefully read all of section 5. Evaluate each one of the examples in
651 section 5 in the MathPiper console and verify that the results match the
652 ones in the book.

653 **5.5.2 Exercise 2**

654 Answer each one of the following questions:

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

656 b) What is a variable?

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

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

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

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

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

662 h) What does the % character do?

663 **5.5.3 Exercise 3**

664 Perform the following calculation:

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

665 **5.5.4 Exercise 4**

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

667 using the following line of code:

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

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

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

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

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

675 In> **answer**

676 **5.5.5 Exercise 5**

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

679 a)

680 In> x

681 b)

682 In> y

683 c)

684 In> z

685 d)

686 In> x + y

687 e)

688 In> x + z

689 f)

690 In> x + y + z

691 **6 The MathPiper Documentation Plugin**

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

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

700 **6.1 Function List**

701 MathPiper's functions are divided into two main categories called **user** functions
702 and **programmer functions**. In general, the **user functions** are used for
703 solving problems in the MathPiper console or with short programs and the
704 **programmer functions** are used for longer programs. However, users will
705 often use some of the programmer functions and programmers will use the user
706 functions as needed.

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

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

724 **6.2 Mini Web Browser Interface**

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

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

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

735 **6.3 Exercises**

736 **6.3.1 Exercise 1**

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

742 **6.3.2 Exercise 2**

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

7 Using MathRider As A Programmer's Text Editor

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

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

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

7.1 Creating, Opening, Saving, And Closing Text Files

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

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

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

7.2 Editing Files

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

7.3 File Modes

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

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

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

791 ***7.4 Learning How To Type Properly Is An Excellent Investment Of Your*** 792 ***Time***

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

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

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

803 ***7.5 Exercises***

804 ***7.5.1 Exercise 1***

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

809 **8 MathRider Worksheet Files**

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

817 **8.1 Code Folds**

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

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

```
828 %mathpiper
829 2 + 3;
830 %/mathpiper
```

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

```
835 %mathpiper
836 2 + 3;
837 %/mathpiper
838     %output,preserve="false"
839     Result: 5
840 .    %/output
```

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

842 %output have their **preserve property** set to **false**. This tells MathRider to
843 overwrite the %output fold with a new version during the next execution of its
844 parent. If preserve is set to **true**, the fold will not be overwritten and a new fold
845 will be created instead.

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

848 8.1.1 The title Attribute

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

```
852 %mathpiper,title="Add two numbers together."
```

```
853 2 + 3;
```

```
854 %/mathpiper
```

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

859 8.2 Automatically Inserting Folds & Removing Unpreserved Folds

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

```
861 %mathpiper
```

```
862 %/mathpiper
```

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

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

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

873 **8.3 *Placing Text Outside Of A Fold***

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

875 Text can be placed above folds like this.

```
876 text text text text
```

```
877 text text text text
```

```
878 %mathpiper,title="Fold 1"
```

```
879 2 + 3;
```

```
880 %/mathpiper
```

881 Text can be placed between folds like this.

```
882 text text text text
```

```
883 text text text text
```

```
884 %mathpiper,title="Fold 2"
```

```
885 3 + 4;
```

```
886 %/mathpiper
```

887 Text can be placed between folds like this.

```
888 text text text text
```

```
889 text text text text
```

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

893 **8.4 *Exercises***

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

896 [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)
897 [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)

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

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

905 **8.4.1 Exercise 1**

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

909 **8.4.2 Exercise 2**

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

912 **8.4.3 Exercise 3**

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

914 **8.4.4 Exercise 4**

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

918 a) $2 * 7 + 3$

919 b) $18 / 3$

920 c) $234238342 + 2038408203$

921 d) $324802984 * 2308098234$

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

9 MathPiper Programming Fundamentals

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

9.1 Values and Expressions

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

3

represents the number three, the value:

0.5

represents the number one half, and the value:

"Mathematics is powerful!"

represents an English sentence.

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

3

2 + 3

5 + 6*21/18 - 2^3

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

In> 2 + 3

Result> 5

9.2 Operators

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

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

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

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

959 In> 5 - 2
960 Result> 3

961 In> 3*4
962 Result> 12

963 In> 30/3
964 Result> 10

965 In> 8%5
966 Result> 3

967 In> 2^3
968 Result> 8

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

970 In> -3
971 Result> -3

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

974 In> - -3
975 Result> 3

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

981 **9.3 Operator Precedence**

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

989 [^] Exponents are evaluated right to left.

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

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

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

995 5 + 6*21/18 - 2^3

996 And here it is in traditional form:

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

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

```
999 5 + 6*21/18 - 2^3
1000 5 + 6*21/18 - 8
1001 5 + 126/18 - 8
1002 5 + 7 - 8
1003 12 - 8
1004 4
```

1005 Starting with the first expression, MathPiper evaluates the [^] operator first which
1006 results in the **8** in the expression below it. In the second expression, the *****
1007 operator is executed next, and so on. The last expression shows that the final
1008 result after all of the operators have been evaluated is **4**.

1009 **9.4 Changing The Order Of Operations In An Expression**

1010 The default order of operations for an expression can be changed by grouping
1011 various parts of the expression within parentheses **()**. Parentheses force the
1012 code that is placed inside of them to be evaluated before any other operators are
1013 evaluated. For example, the expression 2 + 4*5 evaluates to 22 using the
1014 default precedence rules:

```
1015 In> 2 + 4*5
1016 Result> 22
```

1017 If parentheses are placed around 2 + 4, however, the addition operator is forced
1018 to be evaluated before the multiplication operator and the result is 30:

```
1019 In> (2 + 4)*5
1020 Result> 30
```

1021 Parentheses can also be nested and nested parentheses are evaluated from the
1022 most deeply nested parentheses outward:

```
1023 In> ((2 + 4)*3)*5
1024 Result> 90
```

1025 (Note: precedence adjusting parentheses are different from the parentheses that
1026 are used to call functions.)

1027 Since parentheses are evaluated before any other operators, they are placed at
1028 the top of the precedence table:

- 1029 () Parentheses are evaluated from the inside out.
- 1030 ^ Then exponents are evaluated right to left.
- 1031 *,%,/ Then multiplication, remainder, and division operations are evaluated
1032 left to right.
- 1033 +, − Finally, addition and subtraction are evaluated left to right.

1034 **9.5 Functions & Function Names**

1035 In programming, **functions** are named blocks of code that can be executed one
1036 or more times by being **called** from other parts of the same program or called
1037 from other programs. Functions **can have values passed to them** from the
1038 calling code and they **always return a value** back to the calling code when they
1039 are finished executing. An example of a function is the **IsEven()** function which
1040 was discussed in an previous section.

1041 Functions are one way that MathPiper enables code to be reused. Most
1042 programming languages allow code to be reused in this way, although in other
1043 languages these named blocks of code are sometimes called **subroutines**,
1044 **procedures**, or **methods**.

1045 The functions that come with MathPiper have names which consist of either a
1046 single word (such as **Sum()**) or multiple words that have been put together to
1047 form a compound word (such as **IsBound()**). All letters in the names of
1048 functions which come with MathPiper are lower case except the beginning letter
1049 in each word, which are upper case.

1050 **9.6 Functions That Produce Side Effects**

1051 Most functions are executed to obtain the **results** they produce but some
1052 functions are executed in order to **have them perform work that is not in the**
1053 **form of a result**. Functions that perform work that is not in the form of a result
1054 are said to produce **side effects**. Side effects include many forms of work such
1055 as sending information to the user, opening files, and changing values in the
1056 computer's memory.

1057 When a function produces a side effect which sends information to the user, this
1058 information has the words **Side Effects:** placed before it in the output instead of
1059 the word **Result:**. The **Echo()** and **Write()** functions are examples of functions
1060 that produce side effects and they are covered in the next section.

1061 **9.6.1 Printing Related Functions: Echo(), Write(), And Newline()**

1062 The printing related functions send text information to the user and this is
1063 usually referred to as "printing" in this document. However, it may also be called
1064 "echoing" and "writing".

1065 **9.6.1.1 Echo()**

1066 The **Echo()** function takes one expression (or multiple expressions separated by
1067 commas) evaluates each expression, and then prints the results as side effect
1068 output. The following examples illustrate this:

```
1069 In> Echo(1)
1070 Result> True
1071 Side Effects>
1072 1
```

1073 In this example, the number 1 was passed to the Echo() function, the number
1074 was evaluated (all numbers evaluate to themselves), and the result of the
1075 evaluation was then printed as a side effect. Notice that Echo() **also returned a**
1076 **result**. In MathPiper, all functions return a result, but functions whose main
1077 purpose is to produce a side effect usually just return a result of **True** if the side
1078 effect succeeded or **False** if it failed. In this case, Echo() returned a result of
1079 **True** because it was able to successfully print a 1 as its side effect.

1080 The next example shows multiple expressions being sent to Echo() (notice that
1081 the expressions are separated by commas):

```
1082 In> Echo(1,1+2,2*3)
1083 Result> True
1084 Side Effects>
1085 1 3 6
```

1086 The expressions were each evaluated and their results were returned (separated
1087 by spaces) as side effect output. If it is desired that commas be printed between
1088 the numbers in the output, simply place three commas between the expressions
1089 that are passed to Echo():

```
1090 In> Echo(1,,,1+2,,,2*3)
1091 Result> True
1092 Side Effects>
1093 1 , 3 , 6
```

1094 Each time an Echo() function is executed, it always forces the display to drop
1095 down to the next line after it is finished. This can be seen in the following
1096 program which is similar to the previous one except it uses a separate Echo()
1097 function to display each expression:

```
1098 %mathpiper
1099 Echo(1);
1100 Echo(1+2);
1101 Echo(2*3);
1102 %/mathpiper
1103 %output,preserve="false"
1104 Result: True
1105
1106 Side Effects:
1107 1
1108 3
1109 6
1110 . %/output
```

1111 Notice how the 1, the 3, and the 6 are each on their own line.

1112 Now that we have seen how Echo() works, lets use it to do something useful. If
1113 more than one expression is evaluated in a %mathpiper fold, only the result from
1114 the last expression that was evaluated (which is usually the bottommost
1115 expression) is displayed:

```
1116 %mathpiper
1117 a := 1;
1118 b := 2;
1119 c := 3;
1120 %/mathpiper
```

```
1121     %output,preserve="false"
1122     Result: 3
1123 .    %/output
```

1124 In MathPiper, programs are executed one line at a time, starting at the topmost
1125 line of code and working downwards from there. In this example, the line `a := 1;`
1126 is executed first, then the line `b := 2;` is executed, and so on. Notice, however,
1127 that even though we wanted to see what was in all three variables, only the
1128 content of the last variable was displayed.

1129 The following example shows how `Echo()` can be used to display the contents of
1130 all three variables:

```
1131 %mathpiper
1132 a := 1;
1133 Echo(a);
1134 b := 2;
1135 Echo(b);
1136 c := 3;
1137 Echo(c);
1138 %/mathpiper
1139     %output,preserve="false"
1140     Result: True
1141
1142     Side Effects:
1143     1
1144     2
1145     3
1146 .    %/output
```

1147 9.6.1.2 Echo Functions Are Useful For "Debugging" Programs

1148 The errors that are in a program are often called "bugs". This name came from
1149 the days when computers were the size of large rooms and were made using
1150 electromechanical parts. Periodically, bugs would crawl into the machines and
1151 interfere with its moving mechanical parts and this would cause the machine to
1152 malfunction. The bugs needed to be located and removed before the machine
1153 would run properly again.

1154 Of course, even back then most program errors were produced by programmers
1155 entering wrong programs or entering programs wrong, but they liked to say that
1156 all of the errors were caused by bugs and not by themselves! The process of
1157 fixing errors in a program became known as **debugging** and the names "bugs"

1158 and "debugging" are still used by programmers today.

1159 One of the standard ways to locate bugs in a program is to place **Echo()** function
1160 calls in the code at strategic places which **print the contents of variables and**
1161 **display messages**. These Echo() functions will enable you to see what your
1162 program is doing while it is running. After you have found and fixed the bugs in
1163 your program, you can remove the debugging Echo() function calls or comment
1164 them out if you think they may be needed later.

1165 9.6.1.3 Write()

1166 The **Write()** function is similar to the Echo() function except it does not
1167 automatically drop the display down to the next line after it finishes executing:

```
1168 %mathpiper
1169 Write(1,,);
1170 Write(1+2,,);
1171 Echo(2*3);
1172 %/mathpiper
1173     %output,preserve="false"
1174     Result: True
1175
1176     Side Effects:
1177     1,3,6
1178 .    %/output
```

1179 Write() and Echo() have other differences besides the one discussed here and
1180 more information about them can be found in the documentation for these
1181 functions.

1182 9.6.1.4 NewLine()

1183 The **NewLine()** function simply prints a blank line in the side effects output. It
1184 is useful for placing vertical space between printed lines:

```
1185 %mathpiper
1186 a := 1;
1187 Echo(a);
1188 NewLine();
1189
1189 b := 2;
1190 Echo(b);
```



```
1191 NewLine();
1192 c := 3;
1193 Echo(c);
1194 %mathpiper
1195     %output,preserve="false"
1196     Result: True
1197
1198     Side Effects:
1199     1
1200
1201     2
1202
1203     3
1204 . %/output
```

1203 9.7 Expressions Are Separated By Semicolons

1204 As discussed earlier, all of the expressions that are inside of a **%mathpiper** fold
1205 must have a semicolon (;) after them. However, the expressions executed in the
1206 **MathPiper console** did not have a semicolon after them. MathPiper actually
1207 requires that all expressions end with a semicolon, but one does not need to add
1208 a semicolon to an expression which is typed into the MathPiper console **because**
1209 **the console adds it automatically** when the expression is executed.

1210 9.7.1 Placing More Than One Expression On A Line In A Fold

1211 All the previous code examples have had each of their expressions on a separate
1212 line, but multiple expressions can also be placed on a single line because the
1213 semicolons tell MathPiper where one expression ends and the next one begins:

```
1214 %mathpiper
1215 a := 1; Echo(a); b := 2; Echo(b); c := 3; Echo(c);
1216 %/mathpiper
1217     %output,preserve="false"
1218     Result: True
1219
1220     Side Effects:
1221     1
1222     2
1223     3
1224 . %/output
```

1225 The spaces that are in the code of this example are used to make the code more
1226 readable. Any spaces that are present within any expressions or between them
1227 are ignored by MathPiper and if we remove the spaces from the previous code,
1228 the output remains the same:

```
1229 %mathpiper
1230 a:=1;Echo(a);b:=2;Echo(b);c:= 3;Echo(c);
1231 %/mathpiper
1232     %output,preserve="false"
1233     Result: True
1234
1235     Side Effects:
1236     1
1237     2
1238     3
1239 .    %/output
```

1240 9.7.2 Placing Multiple Expressions In A Code Block

1241 A **code block** (which is also called a **compound expression**) consists of one or
1242 more expressions which are separated by semicolons and placed within an open
1243 bracket (**[**) and close bracket (**]**) pair. When a code block is evaluated, each
1244 expression in the block will be executed from left to right. The following
1245 example shows expressions being executed within of a code block inside the
1246 MathPiper console:

```
1247 In> [a := 1; Echo(a); b := 2; Echo(b); c := 3; Echo(c);]
1248 Result> True
1249 Side Effects>
1250 1
1251 2
1252 3
```

1253 Notice that all of the expressions were executed and 1-3 was printed as a side
1254 effect. Code blocks **always return the result of the last expression executed**
1255 **as the result of the whole block**. In this case, **True** was returned as the result
1256 because the last **Echo(c)** function returned **True**. If we place **another**
1257 **expression after the Echo(c) function**, however, **the block will execute this**
1258 **new expression last and its result will be the one returned by the block**:

```
1259 In> [a := 1; Echo(a); b := 2; Echo(b); c := 3; Echo(c); 2 + 2;]
1260 Result> 4
1261 Side Effects>
1262 1
```

1263 2
1264 3

1265 Finally, code blocks can have their contents placed on separate lines if desired:

```
1266 %mathpiper
1267 [
1268     a := 1;
1269
1270     Echo(a);
1271
1272     b := 2;
1273
1274     Echo(b);
1275
1276     c := 3;
1277
1278     Echo(c);
1279 ];
1280
1281 %output,preserve="false"
1282 Result: True
1283
1284 Side Effects:
1285 1
1286 2
1287 3
1288 . %/output
```

1289 Code blocks are very powerful and we will be discussing them further in later
1290 sections.

1291 9.7.2.1 Automatic Bracket, Parentheses, And Brace Match Indicating

1292 In programming, most open brackets '[' have a close bracket ']', most open
1293 parentheses '(' have a close parentheses ')', and most open braces '{' have a
1294 close brace '}'. It is often difficult to make sure that each "open" character has a
1295 matching "close" character and if any of these characters don't have a match,
1296 then an error will be produced.

1297 Thankfully, most programming text editors have a character match indicating
1298 tool that will help locate problems. To try this tool, paste the following code into
1299 a .mrw file and following the directions that are present in its comments:

```
1300 %mathpiper
1301 /*
```

```
1302      Copy this code into a .mrw file. Then, place the cursor
1303      to the immediate right of any {, }, [, ], (, or ) character.
1304      You should notice that the match to this character is
1305      indicated by a rectangle being drawing around it.
1306      */
```

```
1307  list := {1,2,3};
1308  [
1309      Echo("Hello");
1310      Echo(list);
1311  ];
1312  %/mathpiper
```

1313 9.8 Strings

1314 A **string** is a **value** that is used to hold text-based information. The typical
1315 expression that is used to create a string consists of **text which is enclosed**
1316 **within double quotes**. Strings can be assigned to variables just like numbers
1317 can and strings can also be displayed using the Echo() function. The following
1318 program assigns a string value to the variable 'a' and then echos it to the user:

```
1319 %mathpiper
1320 a := "Hello, I am a string.";
1321 Echo(a);
1322 %/mathpiper
1323 %output,preserve="false"
1324 Result: True
1325
1326 Side Effects:
1327 Hello, I am a string.
1328 . %/output
```

1329 9.8.1 The MathPiper Console and MathPiper Folds Can Access The Same 1330 Variables

1331 A useful aspect of using MathPiper inside of MathRider is that variables that are
1332 assigned inside of a **%mathpiper fold** are accessible inside of the **MathPiper**
1333 **console** and variables that are assigned inside of the **MathPiper console** are
1334 available inside of **%mathpiper folds**. For example, after the above fold is
1335 executed, the string that has been bound to variable 'a' can be displayed in the
1336 MathPiper console:

```
1337 In> a
1338 Result> "Hello, I am a string."
```

1339 9.8.2 Using Strings To Make Echo's Output Easier To Read

1340 When the Echo() function is used to print the values of multiple variables, it is
1341 often helpful to print some information next to each variable so that it is easier to
1342 determine which value came from which Echo() function in the code. The
1343 following program prints the name of the variable that each value came from
1344 next to it in the side effects output:

```
1345 %mathpiper
1346 a := 1;
1347 Echo("Variable a: ", a);
1348 b := 2;
1349 Echo("Variable b: ", b);
1350 c := 3;
1351 Echo("Variable c: ", c);
1352 %/mathpiper
1353     %output,preserve="false"
1354     Result: True
1355
1356     Side Effects:
1357     Variable a: 1
1358     Variable b: 2
1359     Variable c: 3
1360 .    %/output
```

1361 9.8.2.1 Combining Strings With The : Operator

1362 If you need to combine two or more strings into one string, you can use the :
1363 operator like this:

```
1364 In> "A" : "B" : "C"
1365 Result: "ABC"
1366 In> "Hello " : "there!"
1367 Result: "Hello there!"
```

1368 9.8.2.2 WriteString()

1369 The **WriteString()** function prints a string without showing the double quotes

1370 that are around it.. For example, here is the Write() function being used to print
1371 the string "Hello":

```
1372 In> Write("Hello")
1373 Result: True
1374 Side Effects:
1375 "Hello"
```

1376 Notice the double quotes? Here is how the WriteString() function prints "Hello":

```
1377 In> WriteString("Hello")
1378 Result: True
1379 Side Effects:
1380 Hello
```

1381 **9.8.2.3 NI()**

1382 The **NI()** (New Line) function is used with the : function to place newline
1383 characters inside of strings:

```
1384 In> WriteString("A": NI() : "B")
1385 Result: True
1386 Side Effects:
1387 A
1388 B
```

1389 **9.8.2.4 Space()**

1390 The Space() function is used to add spaces to printed output:

```
1391 In> WriteString("A"); Space(5); WriteString("B")
1392 Result: True
1393 Side Effects:
1394 A      B
```

```
1395 In> WriteString("A"); Space(10); WriteString("B")
1396 Result: True
1397 Side Effects:
1398 A          B
```

```
1399 In> WriteString("A"); Space(20); WriteString("B")
1400 Result: True
1401 Side Effects:
1402 A                      B
```

1403 **9.8.3 Accessing The Individual Letters In A String**

1404 Individual letters in a string (which are also called **characters**) can be accessed
1405 by placing the character's position number (also called an **index**) inside of

1406 brackets **[]** after the variable it is bound to. A character's position is determined
1407 by its distance from the left side of the string starting at 1. For example, in the
1408 string "Hello", 'H' is at position 1, 'e' is at position 2, etc. The following code
1409 shows individual characters in the above string being accessed:

```
1410 In> a := "Hello, I am a string."  
1411 Result> "Hello, I am a string."
```

```
1412 In> a[1]  
1413 Result> "H"
```

```
1414 In> a[2]  
1415 Result> "e"
```

```
1416 In> a[3]  
1417 Result> "l"
```

```
1418 In> a[4]  
1419 Result> "l"
```

```
1420 In> a[5]  
1421 Result> "o"
```

1422 **9.8.3.1 Indexing Before The Beginning Of A String Or Past The End Of A String**

1423 Lets see what happens if an index is used that is less than **1** or greater than the
1424 length of a given string. First, we will bind the string "Hello" to the variable 'a':

```
1425 In> a := "Hello"  
1426 Result: "Hello"
```

1427 Then, we'll index the character at position **1** and then the character at position **0**:

```
1428 In> a[1]  
1429 Result: "H"
```

```
1430 In> a[0]  
1431 Result:  
1432 Exception: In function "StringMidGet" :  
1433 bad argument number 1(counting from 1) :  
1434 The offending argument aindex evaluated to 0
```

1435 Notice that using an index of **0** resulted in an error.

1436 Next, lets access the character at position **5** (which is the 'o'), then the character
1437 at position **6** and finally the character at position **7**:

```
1438 In> a[5]
```

1439 Result: "o"

1440 In> a[6]

1441 Result: ""

1442 In> a[7]

1443 Result:

1444 Exception: String index out of range: 8

1445 The 'o' at position **5** was returned correctly, but accessing position **6** returned a
1446 double quote character (") and accessing position 7 resulted in an error. What
1447 you can see in this section is that errors are usually produced if an index is not
1448 set to the position of an actual character in a string.

1449 9.9 Comments

1450 Source code can often be difficult to understand and therefore all programming
1451 languages provide the ability for **comments** to be included in the code.

1452 Comments are used to explain what the code near them is doing and they are
1453 usually meant to be read by humans instead of being processed by a computer.
1454 Therefore, comments are ignored by the computer when a program is executed.

1455 There are two ways that MathPiper allows comments to be added to source code.
1456 The first way is by placing two forward slashes // to the left of any text that is
1457 meant to serve as a comment. The text from the slashes to the end of the line
1458 the slashes are on will be treated as a comment. Here is a program that contains
1459 comments which use slashes:

1460 %mathpiper

1461 //This is a comment.

1462 x := 2; //Set the variable x equal to 2.

1463 %/mathpiper

1464 %output,preserve="false"

1465 Result: 2

1466 . %/output

1467 When this program is executed, any text that starts with slashes is ignored.

1468 The second way to add comments to a MathPiper program is by enclosing the
1469 comments inside of slash-asterisk/asterisk-slash symbols /* */. This option is
1470 useful when a comment is too large to fit on one line. Any text between these
1471 symbols is ignored by the computer. This program shows a longer comment
1472 which has been placed between these symbols:


```
1473 %mathpiper
1474 /*
1475  This is a longer comment and it uses
1476  more than one line. The following
1477  code assigns the number 3 to variable
1478  x and then returns it as a result.
1479 */
1480 x := 3;
1481 %/mathpiper
1482     %output,preserve="false"
1483     Result: 3
1484 .    %/output
```

1485 **9.10 How To Tell If MathPiper Has Crashed And What To Do If It Has**

1486 Sometimes code will be evaluated which has one or more unusual errors in it and
1487 the errors will cause MathPiper to "crash". Unfortunately, beginners are more
1488 likely to crash MathPiper than more experienced programmers are because a
1489 beginner's program is more likely to have errors in it. When MathPiper crashes,
1490 no harm is done but it will not work correctly after that. **The only way to
1491 recover from a MathPiper crash is to exit MathRider and then relaunch
1492 it.** All the information in your buffers will be saved and preserved **but the
1493 contents of the console will not be.** Be sure to copy the contents of the
1494 console into a buffer and then save it before restarting.

1495 The main way to tell if MathRider has crashed is that it will indicate that **there
1496 are errors in lines of code that are actually fine.** If you are receiving an
1497 error in code that looks okay to you, simply restarting MathRider may fix the
1498 problem. If you restart MathRider and the error is still present, this usually
1499 means that there really is an error in the code.

1500 **9.11 Exercises**

1501 For the following exercises, create a new MathRider worksheet file called
1502 **book_1_section_9_exercises_<your first name>_<your last name>.mrw.**
1503 **(Note: there are no spaces in this file name).** For example, John Smith's
1504 worksheet would be called:

1505 **book_1_section_9_exercises_john_smith.mrw.**

1506 After this worksheet has been created, place your answer for each exercise that
1507 requires a fold into its own fold in this worksheet. Place a title attribute in the
1508 start tag of each fold which indicates the exercise the fold contains the solution
1509 to. The folds you create should look similar to this one:

```
1510 %mathpiper,title="Exercise 1"
```

```
1511 //Sample fold.
```

```
1512 %/mathpiper
```

1513 If an exercise uses the MathPiper console instead of a fold, copy the work you
1514 did in the console into the worksheet so it can be saved.

1515 9.11.1 Exercise 1

1516 Carefully read all of section 9. Evaluate each one of the examples in
1517 section 9 in the MathPiper worksheet you created or in the MathPiper
1518 console and verify that the results match the ones in the book. Copy all
1519 of the console examples you evaluated into your worksheet so they will be
1520 saved but do not put them in a fold.

1521 9.11.2 Exercise 2

1522 Change the precedence of the following expression using parentheses so that
1523 it prints 20 instead of 14:

```
1524 2 + 3 * 4
```

1525 9.11.3 Exercise 3

1526 Place the following calculations into a fold and then use one Echo()
1527 function per variable to print the results of the calculations. Put
1528 strings in the Echo() functions which indicate which variable each
1529 calculated value is bound to:

```
1530 a := 1+2+3+4+5;
```

```
1531 b := 1-2-3-4-5;
```

```
1532 c := 1*2*3*4*5;
```

```
1533 d := 1/2/3/4/5;
```

1534 9.11.4 Exercise 4

1535 Place the following calculations into a fold and then use one Echo()
1536 function to print the results of all the calculations on a single line
1537 (Remember, the Echo() function can print multiple values if they are
1538 separated by commas.):

```
1539 Clear(x);
```

```
1540 a := 2*2*2*2*2;
```

```
1541 b := 2^5;
```

```
1542 c := x^2 * x^3;
```

```
1543 d := 2^2 * 2^3;
```

1544 9.11.5 Exercise 5

1545 The following code assigns a string which contains all of the upper case
1546 letters of the alphabet to the variable **upper**. Each of the three Echo()
1547 functions prints an index number and the letter that is at that position in
1548 the string. Place this code into a fold and then continue the Echo()
1549 functions so that all 26 letters and their index numbers are printed

```
1550 upper := "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

```
1551 Echo(1,upper[1]);
```

```
1552 Echo(2,upper[2]);
```

```
1553 Echo(3,upper[3]);
```

1554 9.11.6 Exercise 6

1555 Use Echo() functions to print an index number and the character at this
1556 position for the following string (this is similar to what was done in the
1557 previous exercise.):

```
1558 extra := "!.@#$$%^&*() _+<>,?/{ }[]|\-=;";
```

```
1559 Echo(1,extra[1]);
```

```
1560 Echo(2,extra[2]);
```

```
1561 Echo(3,extra[3]);
```

1562 9.11.7 Exercise 7

1563 The following program uses strings and index numbers to print a person's
1564 name. Create a program which uses the three strings from this program to
1565 print the names of three of your favorite musical bands.

```
1566 %mathpiper
```

```
1567 /*
```

```
1568     This program uses strings and index numbers to print
```

```
1569     a person's name.
```

```
1570 */
```

```
1571 upper := "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

```
1572 lower := "abcdefghijklmnopqrstuvwxyz";
```

```
1573 extra := "!.@#$$%^&*() _+<>,?/{ }[]|\-=;";
```

```
1574 //Print "Mary Smith."
```

```
1575 Echo(upper[13],lower[1],lower[18],lower[25],extra[12],upper[19],lower[13],l
```

```
1576 ower[9],lower[20],lower[8],extra[1]);
```

```
1577 %/mathpiper
```

```
1578     %output,preserve="false"
```

```
1579         Result: True
1580
1581         Side Effects:
1582         Mary Smith.
1583     .    %/output
```

1584 10 Rectangular Selection Mode And Text Area Splitting

1585 10.1 Rectangular Selection Mode

1586 One capability that MathRider has that a word processor may not have is the
1587 ability to select rectangular sections of text. To see how this works, do the
1588 following:

- 1589 1) Type three or four lines of text into a text area.
- 1590 2) Hold down the **<Alt>** key then slowly press the **backslash key** (\) a few
1591 times. The bottom of the MathRider window contains a text field which
1592 MathRider uses to communicate information to the user. As **<Alt>** is
1593 repeatedly pressed, messages are displayed which read **Rectangular**
1594 **selection is on** and **Rectangular selection is off**.
- 1595 3) Turn rectangular selection on and then select some text in order to see
1596 how this is different than normal selection mode. **When you are done**
1597 **experimenting, set rectangular selection mode to off.**

1598 Most of the time normal selection mode is what you want to use but in certain
1599 situations rectangular selection mode is better.

1600 10.2 Text area splitting

1601 Sometimes it is useful to have two or more text areas open for a single document
1602 or multiple documents so that different parts of the documents can be edited at
1603 the same time. A situation where this would have been helpful was in the
1604 previous section where the output from an exercise in a MathRider worksheet
1605 contained a list of index numbers and letters which was useful for completing a
1606 later exercise.

1607 MathRider has this ability and it is called **splitting**. If you look just to the right
1608 of the toolbar there is an icon which looks like a blank window, an icon to the
1609 right of it which looks like a window which was split horizontally, and an icon to
1610 the right of the horizontal one which is split vertically. If you let your mouse
1611 hover over these icons, a short description will be displayed for each of them.

1612 Select a text area and then experiment with splitting it by pressing the horizontal
1613 and vertical splitting buttons. Move around these split text areas with their
1614 scroll bars and when you want to unsplit the document, just press the "**Unsplit**
1615 **All**" icon.

1616 10.3 Exercises

1617 For the following exercises, create a new MathRider worksheet file called
1618 **book_1_section_10_exercises_<your first name>_<your last name>.mrw**.

1619 (**Note: there are no spaces in this file name**). For example, John Smith's
1620 worksheet would be called:

1621 **book_1_section_10_exercises_john_smith.mrw.**

1622 For the following exercises, simply type your answers anywhere in the
1623 worksheet.

1624 **10.3.1 Exercise 1**

1625 Carefully read all of section 10 then answer the following questions:

1626 a) Give two examples where rectangular selection mode may be more useful
1627 than regular selection mode.

1628 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
```

```
1669 In> RandomInteger(100)
1670 Result> 82
1671 In> RandomInteger(100)
1672 Result> 93
1673 In> RandomInteger(100)
1674 Result> 32
```

1675 A range of random integers that does not start with 1 can also be generated by
1676 using the **two argument** version of **RandomInteger()**. For example, random
1677 integers between 25 and 75 can be obtained by passing RandomInteger() the
1678 lowest integer in the range and the highest one:

```
1679 In> RandomInteger(25, 75)
1680 Result: 28
1681 In> RandomInteger(25, 75)
1682 Result: 37
1683 In> RandomInteger(25, 75)
1684 Result: 58
1685 In> RandomInteger(25, 75)
1686 Result: 50
1687 In> RandomInteger(25, 75)
1688 Result: 70
```

1689 **11.2 Simulating The Rolling Of Dice**

1690 The following example shows the simulated rolling of a single six sided die using
1691 the RandomInteger() function:

```
1692 In> RandomInteger(6)
1693 Result> 5
1694 In> RandomInteger(6)
1695 Result> 6
1696 In> RandomInteger(6)
1697 Result> 3
1698 In> RandomInteger(6)
1699 Result> 2
1700 In> RandomInteger(6)
1701 Result> 5
```

1702 Code that simulates the rolling of two 6 sided dice can be evaluated in the
1703 MathPiper console by placing it within a **code block**. The following code
1704 outputs the sum of the two simulated dice:

```
1705 In> [a := RandomInteger(6); b := RandomInteger(6); a + b;]
1706 Result> 6
1707 In> [a := RandomInteger(6); b := RandomInteger(6); a + b;]
1708 Result> 12
1709 In> [a := RandomInteger(6); b := RandomInteger(6); a + b;]
1710 Result> 6
```



```
1711 In> [a := RandomInteger(6); b := RandomInteger(6); a + b;]
1712 Result> 4
1713 In> [a := RandomInteger(6); b := RandomInteger(6); a + b;]
1714 Result> 3
1715 In> [a := RandomInteger(6); b := RandomInteger(6); a + b;]
1716 Result> 8
```

1717 Now that we have the ability to simulate the rolling of two 6 sided dice, it would
1718 be interesting to determine if some sums of these dice occur more frequently
1719 than other sums. What we would like to do is to roll these simulated dice
1720 hundreds (or even thousands) of times and then analyze the sums that were
1721 produced. We don't have the programming capability to easily do this yet, but
1722 after we finish the section on **While loops**, we will.

1723 11.3 Exercises

1724 For the following exercises, create a new MathRider worksheet file called
1725 **book_1_section_11_exercises_<your first name>_<your last name>.mrw.**
1726 (**Note: there are no spaces in this file name**). For example, John Smith's
1727 worksheet would be called:

1728 **book_1_section_11_exercises_john_smith.mrw.**

1729 After this worksheet has been created, place your answer for each exercise that
1730 requires a fold into its own fold in this worksheet. Place a title attribute in the
1731 start tag of each fold which indicates the exercise the fold contains the solution
1732 to. The folds you create should look similar to this one:

```
1733 %mathpiper,title="Exercise 1"
1734 //Sample fold.
1735 %/mathpiper
```

1736 If an exercise uses the MathPiper console instead of a fold, copy the work you
1737 did in the console into the worksheet so it can be saved but do not put it in a fold.

1738 11.3.1 Exercise 1

1739 Carefully read all of section 11. Evaluate each one of the examples in
1740 section 11 in the MathPiper worksheet you created or in the MathPiper
1741 console and verify that the results match the ones in the book. Copy all
1742 of the console examples you evaluated into your worksheet so they will be
1743 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
```

```
1764 In> 4 > 5
1765 Result> False
```

```
1766 In> 8 >= 8
1767 Result> True
```

```
1768 In> 5 <= 10
1769 Result> True
```

1770 The following examples show each of the conditional operators in Table 2 being
1771 used to compare values that have been assigned to variables **x** and **y**:

```
1772 %mathpiper
```

```
1773 // Example 1.
1774 x := 2;
1775 y := 3;
```

```
1776 Echo(x, "=", y, ":", x = y);
1777 Echo(x, "!= ", y, ":", x != y);
1778 Echo(x, "< ", y, ":", x < y);
1779 Echo(x, "<= ", y, ":", x <= y);
1780 Echo(x, "> ", y, ":", x > y);
1781 Echo(x, ">= ", y, ":", x >= y);
```

```
1782 %/mathpiper
```

```
1783 %output,preserve="false"
1784 Result: True
1785
1786 Side Effects:
1787 2 = 3 :False
1788 2 != 3 :True
1789 2 < 3 :True
1790 2 <= 3 :True
1791 2 > 3 :False
1792 2 >= 3 :False
1793 . %/output
```

```
1794 %mathpiper
```

```
1795 // Example 2.
1796 x := 2;
1797 y := 2;
```

```
1798 Echo(x, "=", y, ":", x = y);
1799 Echo(x, "!= ", y, ":", x != y);
1800 Echo(x, "< ", y, ":", x < y);
1801 Echo(x, "<= ", y, ":", x <= y);
1802 Echo(x, "> ", y, ":", x > y);
```

```
1803     Echo(x, ">= ", y, ":", x >= y);
```

```
1804 %/mathpiper
```

```
1805     %output,preserve="false"
```

```
1806     Result: True
```

```
1807
```

```
1808     Side Effects:
```

```
1809     2 = 2 :True
```

```
1810     2 != 2 :False
```

```
1811     2 < 2 :False
```

```
1812     2 <= 2 :True
```

```
1813     2 > 2 :False
```

```
1814     2 >= 2 :True
```

```
1815 .    %/output
```

```
1816 %mathpiper
```

```
1817 // Example 3.
```

```
1818 x := 3;
```

```
1819 y := 2;
```

```
1820 Echo(x, "=", y, ":", x = y);
```

```
1821 Echo(x, "!= ", y, ":", x != y);
```

```
1822 Echo(x, "< ", y, ":", x < y);
```

```
1823 Echo(x, "<= ", y, ":", x <= y);
```

```
1824 Echo(x, "> ", y, ":", x > y);
```

```
1825 Echo(x, ">= ", y, ":", x >= y);
```

```
1826 %/mathpiper
```

```
1827     %output,preserve="false"
```

```
1828     Result: True
```

```
1829
```

```
1830     Side Effects:
```

```
1831     3 = 2 :False
```

```
1832     3 != 2 :True
```

```
1833     3 < 2 :False
```

```
1834     3 <= 2 :False
```

```
1835     3 > 2 :True
```

```
1836     3 >= 2 :True
```

```
1837 .    %/output
```

1838 Conditional operators are placed at a lower level of precedence than the other
1839 operators we have covered to this point:

1840 () Parentheses are evaluated from the inside out.

1841 ^ Then exponents are evaluated right to left.

1842 *,%,/ Then multiplication, remainder, and division operations are evaluated
1843 left to right.

1844 +, - Then addition and subtraction are evaluated left to right.

1845 =,!=,<,<=,>,>= Finally, conditional operators are evaluated.

1846 **12.2 Predicate Expressions**

1847 Expressions which return either **True** or **False** are called "**predicate**"
1848 expressions. By themselves, predicate expressions are not very useful and they
1849 only become so when they are used with special decision making functions, like
1850 the If() function (which is discussed in the next section).

1851 **12.3 Exercises**

1852 For the following exercises, create a new MathRider worksheet file called
1853 **book_1_section_12a_exercises_<your first name>_<your last name>.mrw.**
1854 (**Note: there are no spaces in this file name**). For example, John Smith's
1855 worksheet would be called:

1856 **book_1_section_12a_exercises_john_smith.mrw.**

1857 After this worksheet has been created, place your answer for each exercise that
1858 requires a fold into its own fold in this worksheet. Place a title attribute in the
1859 start tag of each fold which indicates the exercise the fold contains the solution
1860 to. The folds you create should look similar to this one:

1861 `%mathpiper,title="Exercise 1"`

1862 `//Sample fold.`

1863 `%/mathpiper`

1864 If an exercise uses the MathPiper console instead of a fold, copy the work you
1865 did in the console into the worksheet so it can be saved but do not put it in a fold.

1866 **12.3.1 Exercise 1**

1867 Carefully read all of section 12 up to this point. Evaluate each one of
1868 the examples in the sections you read in the MathPiper worksheet you
1869 created or in the MathPiper console and verify that the results match the
1870 ones in the book. Copy all of the console examples you evaluated into your
1871 worksheet so they will be saved but do not put them in a fold.

1872 **12.3.2 Exercise 2**

1873 Open a MathPiper session and evaluate the following predicate expressions:

1874 `In> 3 = 3`

1875 `In> 3 = 4`

1876 `In> 3 < 4`

1877 `In> 3 != 4`

1878 `In> -3 < 4`

1879 `In> 4 >= 4`

1880 `In> 1/2 < 1/4`

1881 `In> 15/23 < 122/189`

1882 `/*In the following two expressions, notice that 1/2 is not considered to be`
1883 `equal to .5 unless it is converted to a numerical value first.*/`

1884 `In> 1/2 = .5`

1885 `In> N(1/2) = .5`

1886 **12.3.3 Exercise 3**

1887 Come up with 10 predicate expressions of your own and evaluate them in the
1888 MathPiper console.

1889 **12.4 Making Decisions With The If() Function & Predicate Expressions**

1890 All programming languages have the ability to make decisions and the most
1891 commonly used function for making decisions in MathPiper is the **If()** function.

1892 There are two calling formats for the If() function:

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

1893 The way the first form of the If() function works is that it evaluates the first
1894 expression in its argument list (which is the "**predicate**" expression) and then
1895 looks at the value that is returned. If this value is **True**, the "**then**" expression
1896 that is listed second in the argument list is executed. If the predicate expression
1897 evaluates to **False**, the "**then**" expression is not executed. (Note: any function

1898 that accepts a predicate expression as a parameter can also accept the boolean
1899 values True and False).

1900 The following program uses an **If()** function to determine if the value in variable
1901 **number** is greater than 5. If number is greater than 5, the program will echo
1902 "Greater" and then "End of program":

```
1903 %mathpiper
1904 number := 6;
1905 If(number > 5, Echo(number, "is greater than 5.));
1906 Echo("End of program.");
1907 %/mathpiper
1908 %output,preserve="false"
1909 Result: True
1910
1911 Side Effects:
1912 6 is greater than 5.
1913 End of program.
1914 . %/output
```

1915 In this program, number has been set to 6 and therefore the expression number
1916 > 5 is **True**. When the **If()** function evaluates the **predicate expression** and
1917 determines it is **True**, it then executes the **first Echo()** function. The **second**
1918 **Echo()** function at the bottom of the program prints "End of program"
1919 regardless of what the If() function does. (**Note: semicolons cannot be placed**
1920 **after expressions which are in function calls.**)

1921 Here is the same program except that **number** has been set to **4** instead of **6**:

```
1922 %mathpiper
1923 number := 4;
1924 If(number > 5, Echo(number, "is greater than 5.));
1925 Echo("End of program.");
1926 %/mathpiper
1927 %output,preserve="false"
1928 Result: True
1929
1930 Side Effects:
1931 End of program.
1932 . %/output
```

1933 This time the expression **number > 4** returns a value of **False** which causes the
1934 **If()** function to not execute the "**then**" expression that was passed to it.

1935 12.4.1 If() Functions Which Include An "Else" Parameter

1936 The second form of the If() function takes a third "**else**" expression which is
1937 executed only if the predicate expression is **False**. This program is similar to the
1938 previous one except an "**else**" expression has been added to it:

```
1939 %mathpiper
1940 x := 4;
1941 If(x > 5, Echo(x, "is greater than 5."), Echo(x, "is NOT greater than 5.));
1942 Echo("End of program.");
1943 %/mathpiper
1944     %output, preserve="false"
1945     Result: True
1946
1947     Side Effects:
1948     4 is NOT greater than 5.
1949     End of program.
1950 .    %/output
```

1951 12.5 Exercises

1952 For the following exercises, create a new MathRider worksheet file called
1953 **book_1_section_12b_exercises_<your first name>_<your last name>.mrw**.
1954 (**Note: there are no spaces in this file name**). For example, John Smith's
1955 worksheet would be called:

1956 **book_1_section_12b_exercises_john_smith.mrw**.

1957 After this worksheet has been created, place your answer for each exercise that
1958 requires a fold into its own fold in this worksheet. Place a title attribute in the
1959 start tag of each fold which indicates the exercise the fold contains the solution
1960 to. The folds you create should look similar to this one:

```
1961 %mathpiper, title="Exercise 1"
1962 //Sample fold.
1963 %/mathpiper
```

1964 If an exercise uses the MathPiper console instead of a fold, copy the work you

1965 did in the console into the worksheet so it can be saved but do not put it in a fold.

1966 **12.5.1 Exercise 1**

1967 Carefully read all of section 12 starting at the end of the previous
1968 exercises and up to this point. Evaluate each one of the examples in the
1969 sections you read in the MathPiper worksheet you created or in the
1970 MathPiper console and verify that the results match the ones in the book.
1971 Copy all of the console examples you evaluated into your worksheet so they
1972 will be saved but do not put them in a fold.

1973 **12.5.2 Exercise 2**

1974 Write a program which uses the RandomInteger() function to simulate the
1975 flipping of a coin (Hint: you can use 1 to represent a head and 2 to
1976 represent a tail.). Use predicate expressions, the If() function, and the
1977 Echo() function to print the string "**The coin came up heads.**" or the string
1978 "**The coin came up tails.**", depending on what the simulated coin flip came
1979 up as when the code was executed.

1980 **12.6 The And(), Or(), & Not() Boolean Functions & Infix Notation**

1981 **12.6.1 And()**

1982 Sometimes a programmer needs to check if two or more expressions are all **True**
1983 and one way to do this is with the **And()** function. The And() function has **two**
1984 **calling formats** (or **notations**) and this is the first one:

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

1985 This calling format is able to accept one or more predicate expressions as input.
1986 If **all** of these expressions returns a value of **True**, the And() function will also
1987 return a **True**. However, if **any** of the expressions return a **False**, then the And()
1988 function will return a **False**. This can be seen in the following example:

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

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

```
1993 In> And(False, True)  
1994 Result> False
```

```
1995 In> And(True, True, True, True)  
1996 Result> True
```

```
1997 In> And(True, True, False, True)
1998 Result> False
```

1999 The second format (or notation) that can be used to call the And() function is
2000 called **infix** notation:

```
expression1 And expression2
```

2001 With **infix** notation, an expression is placed on both sides of the And() function
2002 name instead of being placed inside of parentheses that are next to it:

```
2003 In> True And True
2004 Result> True
```

```
2005 In> True And False
2006 Result> False
```

```
2007 In> False And True
2008 Result> False
```

```
2009 In> True And True And True And True
2010 Result: True
```

2011 Infix notation can only accept **two** expressions at a time, but it is often more
2012 convenient to use than function calling notation. The following program also
2013 demonstrates the infix version of the And() function being used:

```
2014 %mathpiper
```

```
2015 a := 7;
2016 b := 9;
```

```
2017 Echo("1: ", a < 5 And b < 10);
2018 Echo("2: ", a > 5 And b > 10);
2019 Echo("3: ", a < 5 And b > 10);
2020 Echo("4: ", a > 5 And b < 10);
```

```
2021 If(a > 5 And b < 10, Echo("These expressions are both true."));
```

```
2022 %/mathpiper
```

```
2023 %output,preserve="false"
2024 Result: True
2025
2026 Side Effects:
2027 1: False
2028 2: False
2029 3: False
```

```
2030         4: True
2031         These expressions are both true.
2032     .    %/output
```

2033 12.6.2 Or()

2034 The Or() function is similar to the And() function in that it has both a function
2035 calling format and an infix calling format and it only works with predicate
2036 expressions. However, instead of requiring that all expressions be **True** in order
2037 to return a **True**, Or() will return a **True** if **one or more expressions are True**.

2038 Here is the function calling format for Or():

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

2039 and this example shows Or() being used with function calling format:

```
2040 In> Or(True, False)
2041 Result> True

2042 In> Or(False, True)
2043 Result> True

2044 In> Or(False, False)
2045 Result> False

2046 In> Or(False, False, False, False)
2047 Result> False

2048 In> Or(False, True, False, False)
2049 Result> True
```

2050 The infix notation format for Or() is as follows:

```
expression1 Or expression2
```

2051 and this example shows infix notation being used:

```
2052 In> True Or False
2053 Result> True

2054 In> False Or True
2055 Result> True

2056 In> False Or False
```

2057 Result> False

2058 The following program also demonstrates the infix version of the Or() function
2059 being used:

2060 %mathpiper

2061 a := 7;

2062 b := 9;

2063 Echo("1: ", a < 5 Or b < 10);

2064 Echo("2: ", a > 5 Or b > 10);

2065 Echo("3: ", a > 5 Or b < 10);

2066 Echo("4: ", a < 5 Or b > 10);

2067 If(a < 5 Or b < 10, Echo("At least one of these expressions is true."));

2068 %/mathpiper

2069 %output,preserve="false"

2070 Result: True

2071

2072 Side Effects:

2073 1: True

2074 2: True

2075 3: True

2076 4: False

2077 At least one of these expressions is true.

2078 . %/output

2079 12.6.3 Not() & Prefix Notation

2080 The **Not()** function works with predicate expressions like the And() and Or()
2081 functions do, except it can only accept **one** expression as input. The way Not()
2082 works is that it changes a **True** value to a **False** value and a **False** value to a
2083 **True** value. Here is the Not()'s function calling format:

```
Not(expression)
```

2084 and this example shows Not() being used with function calling format:

2085 In> Not(True)

2086 Result> False

2087 In> Not(False)

2088 Result> True

2089 Instead of providing an alternative infix calling format like And() and Or() do,
2090 Not()'s second calling format uses **prefix** notation:

```
Not expression
```

2091 Prefix notation looks similar to function notation except no parentheses are used:

```
2092 In> Not True  
2093 Result> False
```

```
2094 In> Not False  
2095 Result> True
```

2096 Finally, here is a program that also uses the prefix version of Not():

```
2097 %mathpiper  
2098 Echo("3 = 3 is ", 3 = 3);  
2099 Echo("Not 3 = 3 is ", Not 3 = 3);  
2100 %/mathpiper  
2101     %output,preserve="false"  
2102     Result: True  
2103  
2104     Side Effects:  
2105     3 = 3 is True  
2106     Not 3 = 3 is False  
2107 .    %/output
```

2108 12.7 Exercises

2109 For the following exercises, create a new MathRider worksheet file called
2110 **book_1_section_12c_exercises_<your first name>_<your last name>.mrw.**
2111 **(Note: there are no spaces in this file name).** For example, John Smith's
2112 worksheet would be called:

2113 **book_1_section_12c_exercises_john_smith.mrw.**

2114 After this worksheet has been created, place your answer for each exercise that
2115 requires a fold into its own fold in this worksheet. Place a title attribute in the
2116 start tag of each fold which indicates the exercise the fold contains the solution
2117 to. The folds you create should look similar to this one:

```
2118 %mathpiper,title="Exercise 1"
```

2119 `//Sample fold.`

2120 `%/mathpiper`

2121 If an exercise uses the MathPiper console instead of a fold, copy the work you
2122 did in the console into the worksheet so it can be saved but do not put it in a fold.

2123 **12.7.1 Exercise 1**

2124 Carefully read all of section 12 starting at the end of the previous
2125 exercises and up to this point. Evaluate each one of the examples in the
2126 sections you read in the MathPiper worksheet you created or in the
2127 MathPiper console and verify that the results match the ones in the book.
2128 Copy all of the console examples you evaluated into your worksheet so they
2129 will be saved but do not put them in a fold.

2130 **12.7.2 Exercise 2**

2131 The following program simulates the rolling of two dice and prints a
2132 message if **both** of the two dice come up less than or equal to 3. Create a
2133 similar program which simulates the flipping of two coins and print the
2134 message "Both coins came up heads." if both coins come up heads.

```
2135 %mathpiper
2136 /*
2137     This program simulates the rolling of two dice and prints a message if
2138     both of the two dice come up less than or equal to 3.
2139 */
```

```
2140 die1 := RandomInteger(6);
2141 die2 := RandomInteger(6);
```

```
2142 Echo("Die1: ", die1, " Die2: ", die2);
2143 NewLine();
```

```
2144 If( die1 <= 3 And die2 <= 3, Echo("Both dice came up <= to 3.") );
```

```
2145 %/mathpiper
```

2146 **12.7.3 Exercise 3**

2147 The following program simulates the rolling of two dice and prints a
2148 message if **either** of the two dice come up less than or equal to 3. Create
2149 a similar program which simulates the flipping of two coins and print the
2150 message "At least one coin came up heads." if at least one coin comes up
2151 heads.

```
2152 %mathpiper
2153 /*
2154     This program simulates the rolling of two dice and prints a message if
2155     either of the two dice come up less than or equal to 3.
```

```
2156 */  
  
2157 die1 := RandomInteger(6);  
2158 die2 := RandomInteger(6);  
  
2159 Echo("Die1: ", die1, " Die2: ", die2);  
2160 NewLine();  
  
2161 If( die1 <= 3 Or die2 <= 3, Echo("At least one die came up <= 3.") );  
  
2162 %/mathpiper
```

2163 13 The While() Looping Function & Bodied Notation

2164 Many kinds of machines, including computers, derive much of their power from
2165 the principle of **repeated cycling**. **Repeated cycling** in a MathPiper program
2166 means to execute one or more expressions over and over again and this process
2167 is called "**looping**". MathPiper provides a number of ways to implement **loops**
2168 in a program and these ways range from straight-forward to subtle.

2169 We will begin discussing looping in MathPiper by starting with the straight-
2170 forward **While** function. The calling format for the **While** function is as follows:

```
2171 While(predicate)  
2172 [  
2173     body_expressions  
2174 ];
```

2175 The **While** function is similar to the **If** function except it will repeatedly execute
2176 the expressions it contains as long as its "predicate" expression is **True**. As soon
2177 as the predicate expression returns a **False**, the While() function skips the
2178 expressions it contains and execution continues with the expression that
2179 immediately follows the While() function (if there is one).

2180 The expressions which are contained in a While() function are called its "**body**"
2181 and all functions which have body expressions are called "**bodied**" functions. If
2182 a body contains more than one expression then these expressions need to be
2183 placed within a **code block** (code blocks were discussed in an earlier section).
2184 What a function's body is will become clearer after studying some example
2185 programs.

2186 13.1 Printing The Integers From 1 to 10

2187 The following program uses a While() function to print the integers from 1 to 10:

```
2188 %mathpiper  
  
2189 // This program prints the integers from 1 to 10.  
  
2190 /*  
2191     Initialize the variable count to 1  
2192     outside of the While "loop".  
2193 */  
2194 count := 1;  
  
2195 While(count <= 10)  
2196 [  
2197     Echo(count);
```



```
2198
2199     count := count + 1; //Increment count by 1.
2200 ];
2201 %/mathpiper
2202     %output,preserve="false"
2203     Result: True
2204
2205     Side Effects:
2206     1
2207     2
2208     3
2209     4
2210     5
2211     6
2212     7
2213     8
2214     9
2215     10
2216 . %/output
```

2217 In this program, a single variable called **count** is created. It is used to tell the
2218 Echo() function which integer to print and it is also used in the predicate
2219 expression that determines if the While() function should continue to **loop** or not.

2220 When the program is executed, **1** is placed into **count** and then the While()
2221 function is called. Notice that **count** is bound to the value 1 **above the While**
2222 **loop**. Setting a variable to an initial value is called **initializing** the variable and
2223 in this case, count needs to be initialized before it is used in the While() function.
2224 The predicate expression **count** <= **10** becomes **1** <= **10** and, since 1 is indeed
2225 less than or equal to 10, a value of **True** is returned by the predicate expression.

2226 The While() function sees that the predicate expression returned a **True** and
2227 therefore it executes all of the expressions inside of its **body** from top to bottom.

2228 The Echo() function prints the current contents of count (which is 1) and then the
2229 expression count := count + 1 is executed.

2230 The expression **count** := **count** + **1** is a standard expression form that is used in
2231 many programming languages. Each time an expression in this form is
2232 evaluated, it **increases the variable it contains by 1**. Another way to describe
2233 the effect this expression has on **count** is to say that it **increments count by 1**.

2234 In this case **count** contains **1** and, after the expression is evaluated, **count**
2235 contains **2**.

2236 After the last expression inside the body of the While() function is executed, the
2237 While() function reevaluates its predicate expression to determine whether it
2238 should continue looping or not. Since **count** is **2** at this point, the predicate
2239 expression returns **True** and the code inside the body of the While() function is

2240 executed again. This loop will be repeated until **count** is incremented to **11** and
2241 the predicate expression returns **False**.

2242 **13.2 Printing The Integers From 1 to 100**

2243 The previous program can be adjusted in a number of ways to achieve different
2244 results. For example, the following program prints the integers from 1 to 100 by
2245 changing the **10** in the predicate expression to **100**. A Write() function is used in
2246 this program so that its output is displayed on the same line until it encounters
2247 the **wrap margin** in MathRider (which can be set in Utilities -> Buffer
2248 Options...).

```
2249 %mathpiper
2250 // Print the integers from 1 to 100.
2251 count := 1;
2252 While(count <= 100)
2253 [
2254     Write(count,,);
2255     count := count + 1; //Increment count by 1.
2256 ];
2257
2258 %/mathpiper
2259     %output,preserve="false"
2260     Result: True
2261
2262     Side Effects:
2263     1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,
2264     24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,
2265     44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,
2266     64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,
2267     84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100
2268 . %/output
```

2269 (Note: In MathRider, the above numbers will all be on a single line.)

2270 **13.3 Printing The Odd Integers From 1 To 99**

2271 The following program prints the odd integers from 1 to 99 by changing the
2272 **increment value** in the increment expression from **1** to **2**:

```
2273 %mathpiper
2274 //Print the odd integers from 1 to 99.
```

```
2275 x := 1;
2276 While(x <= 100)
2277 [
2278     Write(x,,);
2279     x := x + 2;    //Increment x by 2.
2280 ];
2281 %/mathpiper
2282     %output,preserve="false"
2283     Result: True
2284
2285     Side Effects:
2286     1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39,41,43,
2287     45,47,49,51,53,55,57,59,61,63,65,67,69,71,73,75,77,79,81,83,
2288     85,87,89,91,93,95,97,99
2289 .    %/output
```

2290 **13.4 Printing The Integers From 1 To 100 In Reverse Order**

2291 Finally, the following program prints the integers from 1 to 100 in reverse order:

```
2292 %mathpiper
2293 //Print the integers from 1 to 100 in reverse order.
2294 x := 100;
2295 While(x >= 1)
2296 [
2297     Write(x,,);
2298     x := x - 1;    //Decrement x by 1.
2299 ];
2300 %/mathpiper
2301     %output,preserve="false"
2302     Result: True
2303
2304     Side Effects:
2305     100,99,98,97,96,95,94,93,92,91,90,89,88,87,86,85,84,83,82,
2306     81,80,79,78,77,76,75,74,73,72,71,70,69,68,67,66,65,64,63,
2307     62,61,60,59,58,57,56,55,54,53,52,51,50,49,48,47,46,45,44,
2308     43,42,41,40,39,38,37,36,35,34,33,32,31,30,29,28,27,26,25,
2309     24,23,22,21,20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,
2310     3,2,1
2311 .    %/output
```

2312 In order to achieve the reverse ordering, this program had to initialize **x** to **100**,
2313 check to see if **x** was **greater than or equal to 1** ($x \geq 1$), and **decrement** **x** by
2314 **subtracting 1 from it** instead of adding 1 to it.

2315 **13.5 Expressions Inside Of Code Blocks Are Indented**

2316 In the programs in the previous sections which use While loops, notice that the
2317 expressions which are inside of the While() function's code block are **indented**.
2318 These expressions do not need to be indented to execute properly, but doing so
2319 makes the program easier to read.

2320 **13.6 Long-Running Loops, Infinite Loops, & Interrupting Execution**

2321 It is easy to create a loop that will execute a **large number of times**, or even **an**
2322 **infinite number of times**, either on purpose or by mistake. When you execute
2323 a program that contains an **infinite loop**, it will run until you tell MathPiper to
2324 **interrupt** its execution. This is done by opening the MathPiper **console** and
2325 then pressing the "**Halt Calculation**" button which in the upper left corner of
2326 the console.

2327 Lets experiment with the **Halt Calculation** button by executing a program that
2328 contains an infinite loop and then stopping it:

```
2329 %mathpiper
2330 //Infinite loop example program.
2331 x := 1;
2332 While(x < 10)
2333 [
2334     x := 3; //Oops, x is not being incremented!.
2335 ];
2336 %/mathpiper
2337     %output,preserve="false"
2338     Processing...
2339 . %/output
```

2340 Since the contents of **x** is never changed inside the loop, the expression **x < 10**
2341 always evaluates to **True** which causes the loop to continue looping. Notice that
2342 the %output fold contains the word "**Processing...**" to indicate that the program
2343 is still running the code.

2344 Execute this program now and then interrupt it using the **Halt Calculation**
2345 button. When the program is interrupted, the %output fold will display the
2346 message "**User interrupted calculation**" to indicate that the program was

2347 interrupted. After a program has been interrupted, the program can be edited
2348 and then rerun.

2349 **13.7 A Program That Simulates Rolling Two Dice 50 Times**

2350 The following program is larger than the previous programs that have been
2351 discussed in this book, but it is also more interesting and more useful. It uses a
2352 While() loop to simulate the rolling of two dice 50 times and it records how many
2353 times each possible sum has been rolled so that this data can be printed. The
2354 comments in the code explain what each part of the program does. (Remember, if
2355 you copy this program to a MathRider worksheet, you can use **rectangular**
2356 **selection mode** to easily remove the line numbers).

```
2357 %mathpiper
2358 /*
2359     This program simulates rolling two dice 50 times.
2360 */

2361 /*
2362     These variables are used to record how many times
2363     a possible sum of two dice has been rolled. They are
2364     all initialized to 0 before the simulation begins.
2365 */
2366 numberOfTwosRolled := 0;
2367 numberOfThreesRolled := 0;
2368 numberOfFoursRolled := 0;
2369 numberOfFivesRolled := 0;
2370 numberOfSixesRolled := 0;
2371 numberOfSevensRolled := 0;
2372 numberOfEightsRolled := 0;
2373 numberOfNinesRolled := 0;
2374 numberOfTensRolled := 0;
2375 numberOfElevensRolled := 0;
2376 numberOfTwelvesRolled := 0;

2377 //This variable keeps track of the number of the current roll.
2378 roll := 1;

2379 Echo("These are the rolls:");

2380 /*
2381     The simulation is performed inside of this While loop. The number of
2382     times the dice will be rolled can be changed by changing the number 50
2383     which is in the While function's predicate expression.
2384 */
2385 While(roll <= 50)
```

```
2386 [
2387     //Roll the dice.
2388     die1 := RandomInteger(6);
2389     die2 := RandomInteger(6);
2390
2391
2392     //Calculate the sum of the two dice.
2393     rollSum := die1 + die2;
2394
2395
2396     /*
2397     Print the sum that was rolled. Note: if a large number of rolls
2398     is going to be performed (say > 1000), it would be best to comment
2399     out this Write() function so that it does not put too much text
2400     into the output fold.
2401     */
2402     Write(rollSum,,);
2403
2404
2405     /*
2406     These If() functions determine which sum was rolled and then add
2407     1 to the variable which is keeping track of the number of times
2408     that sum was rolled.
2409     */
2410     If(rollSum = 2, numberOfTwosRolled := numberOfTwosRolled + 1);
2411     If(rollSum = 3, numberOfThreesRolled := numberOfThreesRolled + 1);
2412     If(rollSum = 4, numberOfFoursRolled := numberOfFoursRolled + 1);
2413     If(rollSum = 5, numberOfFivesRolled := numberOfFivesRolled + 1);
2414     If(rollSum = 6, numberOfSixesRolled := numberOfSixesRolled + 1);
2415     If(rollSum = 7, numberOfSevensRolled := numberOfSevensRolled + 1);
2416     If(rollSum = 8, numberOfEightsRolled := numberOfEightsRolled + 1);
2417     If(rollSum = 9, numberOfNinesRolled := numberOfNinesRolled + 1);
2418     If(rollSum = 10, numberOfTensRolled := numberOfTensRolled + 1);
2419     If(rollSum = 11, numberOfElevensRolled := numberOfElevensRolled+1);
2420     If(rollSum = 12, numberOfTwelvesRolled := numberOfTwelvesRolled+1);
2421
2422
2423     //Increment the roll variable to the next roll number.
2424     roll := roll + 1;
2425 ];

```

```
2426 //Print the contents of the sum count variables for visual analysis.
2427 NewLine();
2428 NewLine();
2429 Echo("Number of Twos rolled: ", numberOfTwosRolled);
2430 Echo("Number of Threes rolled: ", numberOfThreesRolled);
2431 Echo("Number of Fours rolled: ", numberOfFoursRolled);
2432 Echo("Number of Fives rolled: ", numberOfFivesRolled);
2433 Echo("Number of Sixes rolled: ", numberOfSixesRolled);
2434 Echo("Number of Sevens rolled: ", numberOfSevensRolled);
2435 Echo("Number of Eights rolled: ", numberOfEightsRolled);
```

```
2436 Echo("Number of Nines rolled: ", numberOfNinesRolled);
2437 Echo("Number of Tens rolled: ", numberOfTensRolled);
2438 Echo("Number of Elevens rolled: ", numberOfElevensRolled);
2439 Echo("Number of Twelves rolled: ", numberOfTwelvesRolled);

2440 %/mathpiper

2441 %output,preserve="false"
2442 Result: True
2443
2444 Side effects:
2445 These are the rolls:
2446 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,
2447 12,7,8,12,6,8,10,10,5,9,8,4,5,3,5,7,7,4,6,7,7,5,8,
2448
2449 Number of Twos rolled: 0
2450 Number of Threes rolled: 3
2451 Number of Fours rolled: 6
2452 Number of Fives rolled: 4
2453 Number of Sixes rolled: 6
2454 Number of Sevens rolled: 13
2455 Number of Eights rolled: 6
2456 Number of Nines rolled: 3
2457 Number of Tens rolled: 2
2458 Number of Elevens rolled: 4
2459 Number of Twelves rolled: 3
2460 . %/output
```

2461 13.8 Exercises

2462 For the following exercises, create a new MathRider worksheet file called
2463 **book_1_section_13_exercises_<your first name>_<your last name>.mrw.**
2464 (**Note: there are no spaces in this file name**). For example, John Smith's
2465 worksheet would be called:

2466 **book_1_section_13_exercises_john_smith.mrw.**

2467 After this worksheet has been created, place your answer for each exercise that
2468 requires a fold into its own fold in this worksheet. Place a title attribute in the
2469 start tag of each fold which indicates the exercise the fold contains the solution
2470 to. The folds you create should look similar to this one:

```
2471 %mathpiper,title="Exercise 1"

2472 //Sample fold.

2473 %/mathpiper
```

2474 If an exercise uses the MathPiper console instead of a fold, copy the work you
2475 did in the console into the worksheet so it can be saved but do not put it in a fold.

2476 **13.8.1 Exercise 1**

2477 Carefully read all of section 13 up to this point. Evaluate each one of
2478 the examples in the sections you read in the MathPiper worksheet you
2479 created or in the MathPiper console and verify that the results match the
2480 ones in the book. Copy all of the console examples you evaluated into your
2481 worksheet so they will be saved but do not put them in a fold.

2482 **13.8.2 Exercise 2**

2483 Create a program which uses a While loop to print the even integers from 2
2484 to 50 inclusive.

2485 **13.8.3 Exercise 3**

2486 Create a program which prints all the multiples of 5 between 5 and 50
2487 inclusive.

2488 **13.8.4 Exercise 4**

2489 Create a program which simulates the flipping of a **single coin** 500 times.
2490 Print the number of times the coin came up heads and the number of times it
2491 came up tails after the loop is finished executing.

2492 14 Predicate Functions

2493 A **predicate function** is a function that either returns **True** or **False**. Most
2494 predicate functions in MathPiper have names which begin with "**Is**". For
2495 example, **IsEven()**, **IsOdd()**, **IsInteger()**, etc. The following examples show
2496 some of the predicate functions that are in MathPiper:

```
2497 In> IsEven(4)
2498 Result> True
```

```
2499 In> IsEven(5)
2500 Result> False
```

```
2501 In> IsZero(0)
2502 Result> True
```

```
2503 In> IsZero(1)
2504 Result> False
```

```
2505 In> IsNegativeInteger(-1)
2506 Result> True
```

```
2507 In> IsNegativeInteger(1)
2508 Result> False
```

```
2509 In> IsPrime(7)
2510 Result> True
```

```
2511 In> IsPrime(100)
2512 Result> False
```

2513 There is also an **IsBound()** predicate function that can be used to determine
2514 whether or not a value is bound to a given variable:

```
2515 In> a
2516 Result> a
```

```
2517 In> IsBound(a)
2518 Result> False
```

```
2519 In> a := 1
2520 Result> 1
```

```
2521 In> IsBound(a)
2522 Result> True
```

```
2523 In> Clear(a)
2524 Result> True
```

```
2525 In> a
2526 Result> a
```

```
2527 In> IsBound(a)
2528 Result> False
```

2529 The complete list of predicate functions is contained in the **User**
2530 **Functions/Predicates** node in the MathPiperDocs plugin.

2531 **14.1 Finding Prime Numbers With A Loop**

2532 Predicate functions are very powerful when they are combined with loops
2533 because they can be used to automatically make numerous checks. The
2534 following program uses a While loop to pass the integers 1 through 20 (one at a
2535 time) to the **IsPrime()** function in order to determine which integers are prime
2536 and which integers are not prime:

```
2537 %mathpiper
2538 //Determine which numbers between 1 and 20 (inclusive) are prime.
2539 x := 1;
2540 While(x <= 20)
2541 [
2542     primeStatus := IsPrime(x);
2543     Echo(x, "is prime: ", primeStatus);
2544     x := x + 1;
2545 ];
2546
2547 %/mathpiper
2548
2549 %output,preserve="false"
2550 Result: True
2551
2552 Side Effects:
2553 1 is prime: False
2554 2 is prime: True
2555 3 is prime: True
2556 4 is prime: False
2557 5 is prime: True
2558 6 is prime: False
2559 7 is prime: True
2560 8 is prime: False
2561 9 is prime: False
2562 10 is prime: False
2563 11 is prime: True
2564 12 is prime: False
```

```
2565         13 is prime: True
2566         14 is prime: False
2567         15 is prime: False
2568         16 is prime: False
2569         17 is prime: True
2570         18 is prime: False
2571         19 is prime: True
2572         20 is prime: False
2573     .    %/output
```

2574 This program worked fairly well, but it is limited because it prints a line for each
2575 prime number and also each non-prime number. This means that if large ranges
2576 of integers were processed, enormous amounts of output would be produced.
2577 The following program solves this problem by using an If() function to only print
2578 a number if it is prime:

```
2579 %mathpiper
2580 //Print the prime numbers between 1 and 50 (inclusive).
2581 x := 1;
2582 While(x <= 50)
2583 [
2584     primeStatus := IsPrime(x);
2585
2586     If(primeStatus = True, Write(x,,) );
2587
2588     x := x + 1;
2589 ];
2590 %/mathpiper
2591     %output,preserve="false"
2592     Result: True
2593
2594     Side Effects:
2595     2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,
2596 .    %/output
```

2597 This program is able to process a much larger range of numbers than the
2598 previous one without having its output fill up the text area. However, the
2599 program itself can be shortened by moving the **IsPrime()** function **inside** of the
2600 **If()** function instead of using the **primeStatus** variable to communicate with it:

```
2601 %mathpiper
2602 /*
```

```
2603     Print the prime numbers between 1 and 50 (inclusive).
2604     This is a shorter version which places the IsPrime() function
2605     inside of the If() function instead of using a variable.
2606 */

2607 x := 1;

2608 While(x <= 50)
2609 [
2610     If(IsPrime(x), Write(x,,) );
2611
2612     x := x + 1;
2613 ];

2614 %/mathpiper

2615     %output,preserve="false"
2616     Result: True
2617
2618     Side Effects:
2619     2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,
2620 .    %/output
```

2621 14.2 Finding The Length Of A String With The Length() Function

2622 Strings can contain zero or more characters and the **Length()** function can be
2623 used to determine how many characters a string holds:

```
2624 In> s := "Red"
2625 Result> "Red"

2626 In> Length(s)
2627 Result> 3
```

2628 In this example, the string "Red" is assigned to the variable **s** and then **s** is
2629 passed to the **Length()** function. The **Length()** function returned a **3** which
2630 means the string contained **3 characters**.

2631 The following example shows that strings can also be passed to functions
2632 directly:

```
2633 In> Length("Red")
2634 Result> 3
```

2635 An **empty string** is represented by **two double quote marks with no space in**
2636 **between them**. The **length** of an empty string is **0**:

```
2637 In> Length("")
2638 Result> 0
```

2639 **14.3 Converting Numbers To Strings With The String() Function**

2640 Sometimes it is useful to convert a number to a string so that the individual
2641 digits in the number can be analyzed or manipulated. The following example
2642 shows a **number** being converted to a **string** with the **String()** function so that
2643 its **leftmost** and **rightmost** digits can be placed into **variables**:

```
2644 In> number := 523
2645 Result> 523
```

```
2646 In> stringNumber := String(number)
2647 Result> "523"
```

```
2648 In> leftmostDigit := stringNumber[1]
2649 Result> "5"
```

```
2650 In> rightmostDigit := stringNumber[ Length(stringNumber) ]
2651 Result> "3"
```

2652 Notice that the Length() function is used here to determine which character in
2653 **stringNumber** held the **rightmost** digit. Also, keep in mind that when numbers
2654 are in string form, operations such as +, -, *, and / **cannot** be performed on
2655 them.

2656 **14.4 Finding Prime Numbers Which End With 7 (And Multi-line Function Calls)**

2658 Now that we have covered how to turn a number into a string, lets use this
2659 ability inside a loop. The following program finds all the **prime numbers**
2660 between **1** and **500** which have a **7 as their rightmost digit**. There are three
2661 important things which are shown in this program:

2662 1) Function calls **can have their parameters placed on more than one**
2663 **line** if the parameters are too long to fit on a **single line**. In this case, a long
2664 code block is being placed inside of an If() function.

2665 2) Code blocks (which are considered to be compound expressions) **cannot**
2666 **have a semicolon placed after them if they are in a function call**. If a
2667 semicolon is placed after this code block, an error will be produced.

2668 3) If() functions can be placed inside of other If() functions in order to make
2669 more complex decisions. This is referred to as **nesting** functions.

2670 When the program is executed, it finds 24 prime numbers which have 7 as their

2671 rightmost digit:

```
2672 %mathpiper
2673 /*
2674     Find all the prime numbers between 1 and 500 which have a 7
2675     as their rightmost digit.
2676 */
2677 x := 1;
2678 While(x <= 500)
2679 [
2680     //Notice how function parameters can be put on more than one line.
2681     If(IsPrime(x),
2682         [
2683             stringVersionOfNumber := String(x);
2684             stringLength := Length(stringVersionOfNumber);
2685             //Notice that If() functions can be placed inside of other
2686             // If() functions.
2687             If(stringVersionOfNumber[stringLength] = "7", Write(x,,) );
2688         ] //Notice that semicolons cannot be placed after code blocks
2689         //which are in function calls.
2690     ); //This is the close parentheses for the outer If() function.
2691     x := x + 1;
2692 ];
2693
2694 %/mathpiper
2695
2696 %output,preserve="false"
2697 Result: True
2698
2699 Side Effects:
2700 7,17,37,47,67,97,107,127,137,157,167,197,227,257,277,307,317,
2701 337,347,367,397,457,467,487,
2702 . %/output
```

2706 It would be nice if we had the ability to store these numbers someplace so that
2707 they could be processed further and this is discussed in the next section.

2708 14.5 Exercises

2709 For the following exercises, create a new MathRider worksheet file called
2710 **book_1_section_14_exercises_<your first name>_<your last name>.mrw.**
2711 (**Note: there are no spaces in this file name**). For example, John Smith's

2712 worksheet would be called:

2713 **book_1_section_14_exercises_john_smith.mrw.**

2714 After this worksheet has been created, place your answer for each exercise that
2715 requires a fold into its own fold in this worksheet. Place a title attribute in the
2716 start tag of each fold which indicates the exercise the fold contains the solution
2717 to. The folds you create should look similar to this one:

2718 `%mathpiper,title="Exercise 1"`

2719 `//Sample fold.`

2720 `%/mathpiper`

2721 If an exercise uses the MathPiper console instead of a fold, copy the work you
2722 did in the console into the worksheet so it can be saved but do not put it in a fold.

2723 **14.5.1 Exercise 1**

2724 Write a program which uses a While loop to determine how many prime numbers
2725 there are between 1 and 1000. You do not need to print the numbers
2726 themselves, just how many there are.

2727 **14.5.2 Exercise 2**

2728 Write a program which uses a While loop to print all of the prime numbers
2729 between 10 and 99 which contain the digit 3 in either their 1's place or
2730 their 10's place.

2731 15 Lists: Values That Hold Sequences Of Expressions

2732 The **list** value type is designed to hold expressions in an **ordered collection** or
2733 **sequence**. Lists are very flexible and they are one of the most heavily used
2734 value types in MathPiper. Lists can **hold expressions of any type**, they can be
2735 made to **grow and shrink as needed**, and they can be **nested**. Expressions in a
2736 list can be **accessed by their position** in the list (similar to the way that
2737 characters in a string are accessed) and they can also be **replaced by other**
2738 **expressions**.

2739 One way to create a list is by placing zero or more expressions separated by
2740 commas inside of a **pair of braces {}**. In the following example, a list is created
2741 that contains various expressions and then it is assigned to the variable **x**:

```
2742 In> exampleList := {7,42,"Hello",1/2,var}  
2743 Result> {7,42,"Hello",1/2,var}
```

```
2744 In> exampleList  
2745 Result> {7,42,"Hello",1/2,var}
```

2746 The number of expressions in a list can be determined with the **Length()**
2747 function:

```
2748 In> Length({7,42,"Hello",1/2,var})  
2749 Result> 5
```

2750 A single expression in a list can be accessed by placing a set of **brackets []** to
2751 the right of the variable that is bound to the list and then putting the
2752 expression's position number inside of the brackets (**Note: the first expression**
2753 **in the list is at position 1 counting from the left end of the list**):

```
2754 In> exampleList[1]  
2755 Result> 7
```

```
2756 In> exampleList[2]  
2757 Result> 42
```

```
2758 In> exampleList[3]  
2759 Result> "Hello"
```

```
2760 In> exampleList[4]  
2761 Result> 1/2
```

```
2762 In> exampleList[5]  
2763 Result> var
```

2764 The **1st** and **2nd** expressions in this list are **integers**, the **3rd** expression is a

2765 **string**, the **4th** expression is a **rational number** and the **5th** expression is an
2766 **unbound variable**.

2767 Lists can also hold other lists as shown in the following example:

```
2768 In> exampleList := {20, 30, {31, 32, 33}, 40}
```

```
2769 Result> {20,30,{31,32,33},40}
```

```
2770 In> exampleList[1]
```

```
2771 Result> 20
```

```
2772 In> exampleList[2]
```

```
2773 Result> 30
```

```
2774 In> exampleList[3]
```

```
2775 Result> {31,32,33}
```

```
2776 In> exampleList[4]
```

```
2777 Result> 40
```

```
2778
```

2779 The expression in the **3rd** position in the list is another **list** which contains the
2780 integers **31**, **32**, and **33**.

2781 An expression in this second list can be accessed by two **two sets of brackets**:

```
2782 In> exampleList[3][2]
```

```
2783 Result> 32
```

2784 The **3** inside of the first set of brackets accesses the **3rd** member of the **first** list
2785 and the **2** inside of the second set of brackets accesses the **2nd** member of the
2786 **second** list.

2787 **15.1 Append() & Nondestructive List Operations**

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

2788 The **Append()** function adds an expression to the end of a list:

```
2789 In> testList := {21,22,23}
```

```
2790 Result> {21,22,23}
```

```
2791 In> Append(testList, 24)
```

```
2792 Result> {21,22,23,24}
```

2793 However, instead of changing the **original** list, **Append()** creates a **copy** of the
2794 **original** list and appends the expression to the **copy**. This can be confirmed by
2795 evaluating the variable **testList** after the **Append()** function has been called:

```
2796 In> testList
2797 Result> {21,22,23}
```

2798 Notice that the list that is bound to **testList** was not modified by the **Append()**
2799 function. This is called a **nondestructive list operation** and **most MathPiper**
2800 **functions that manipulate lists do so nondestructively**. To have the new list
2801 bound to the variable that is being used, the following technique can be
2802 employed:

```
2803 In> testList := {21,22,23}
2804 Result> {21,22,23}

2805 In> testList := Append(testList, 24)
2806 Result> {21,22,23,24}

2807 In> testList
2808 Result> {21,22,23,24}
```

2809 After this code has been executed, the new list has indeed been bound to
2810 **testList** as desired.

2811 There are some functions, such as **DestructiveAppend()**, which **do** change the
2812 original list and most of them begin with the word "Destructive". These are
2813 called "destructive functions" and they are advanced functions which are not
2814 covered in this book.

2815 **15.2 Using While Loops With Lists**

2816 Functions that loop can be used to **select each expression in a list in turn** so
2817 that an operation can be performed on these expressions. The following
2818 program uses a While loop to print each of the expressions in a list:

```
2819 %mathpiper
2820 //Print each number in the list.
2821 testList := {55,93,40,21,7,24,15,14,82};
2822 index := 1;
2823 While(index <= Length(testList))
2824 [
2825     Echo(index, "- ", testList[index]);
2826     index := index + 1;
2827 ];
2828 %/mathpiper
```

```
2829 %output,preserve="false"
2830 Result: True
2831
2832 Side Effects:
2833 1 - 55
2834 2 - 93
2835 3 - 40
2836 4 - 21
2837 5 - 7
2838 6 - 24
2839 7 - 15
2840 8 - 14
2841 9 - 82
2842 . %/output
```

2843 A **loop** can also be used to search through a list. The following program uses a
2844 **While()** function and an **If()** function to search through a list to see if it contains
2845 the number **53**. If 53 is found in the list, a message is printed:

```
2846 %mathpiper
2847 //Determine if 53 is in the list.
2848 testList := {18,26,32,42,53,43,54,6,97,41};
2849 index := 1;
2850 While(index <= Length(testList))
2851 [
2852     If(testList[index] = 53,
2853         Echo("53 was found in the list at position", index));
2854     index := index + 1;
2855 ];
2856
2857 %/mathpiper
2858 %output,preserve="false"
2859 Result: True
2860
2861 Side Effects:
2862 53 was found in the list at position 5
2863 . %/output
```

2864 When this program was executed, it determined that **53** was present in the list at
2865 position **5**.

2866 15.2.1 Using A While Loop And Append() To Place Values Into A List

2867 In an earlier section it was mentioned that it would be nice if we could store a set
2868 of values for later processing and this can be done with a **While loop** and the
2869 **Append()** function. The following program creates an empty list and assigned it
2870 to the variable **primes**. The **While loop** and the **IsPrime()** function is then used
2871 to locate the prime integers between 1 and 50 and the **Append()** function is used
2872 to place them in the list. The last part of the program then prints some
2873 information about the numbers that were placed into the list:

```
2874 %mathpiper
2875 //Place the prime numbers between 1 and 50 (inclusive) into a list.
2876 //Create an empty list.
2877 primesList := {};
2878 index := 1;
2879 While(index <= 50)
2880 [
2881     /*
2882         If x is prime, append it to the end of the list and then assign
2883         the new list that is created to the variable 'primes'.
2884     */
2885     If(IsPrime(index), primesList := Append(primesList, index ) );
2886     index := index + 1;
2887 ];
2888
2889 //Print information about the primes that were found.
2890 Echo("Primes ", primesList);
2891 Echo("The number of primes in the list = ", Length(primesList) );
2892 Echo("The first number in the list = ", primesList[1] );
2893 %/mathpiper
2894 %output,preserve="false"
2895 Result: True
2896
2897 Side Effects:
2898 Primes {2,3,5,7,11,13,17,19,23,29,31,37,41,43,47}
2899 The number of primes in the list = 15
2900 The first number in the list = 2
2901 . %/output
```

2902 The ability to place values into a list with a loop is very powerful and we will be
2903 using this ability throughout the rest of the book.

2904 **15.3 Exercises**

2905 For the following exercises, create a new MathRider worksheet file called
2906 **book_1_section_15a_exercises_<your first name>_<your last name>.mrw.**
2907 **(Note: there are no spaces in this file name).** For example, John Smith's
2908 worksheet would be called:

2909 **book_1_section_15a_exercises_john_smith.mrw.**

2910 After this worksheet has been created, place your answer for each exercise that
2911 requires a fold into its own fold in this worksheet. Place a title attribute in the
2912 start tag of each fold which indicates the exercise the fold contains the solution
2913 to. The folds you create should look similar to this one:

2914 `%mathpiper,title="Exercise 1"`

2915 `//Sample fold.`

2916 `%/mathpiper`

2917 If an exercise uses the MathPiper console instead of a fold, copy the work you
2918 did in the console into the worksheet so it can be saved but do not put it in a fold.

2919 **15.3.1 Exercise 1**

2920 Create a program that uses a While loop and an IsOdd() predicate function
2921 to analyze the following list and then print the number of odd numbers it
2922 contains.

2923 `{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}`

2924 **15.3.2 Exercise 2**

2925 Create a program that uses a While loop and an IsNegativeNumber() function
2926 to copy all of the negative numbers in the following list into a new list.
2927 Use the variable **negativeNumbersList** to hold the new list. Print the
2928 contents of the list after it has been created.

2929 `{36,-29,-33,-6,14,7,-16,-3,-14,37,-38,-8,-45,-21,-26,6,6,38,-20,33,41,-`
2930 `4,24,37,40,29}`

2931 **15.3.3 Exercise 3**

2932 Create one program that uses a single While loop to analyze the following
2933 list and then print the following information about it:

- 2934 1) The largest number in the list.
2935 2) The smallest number in the list.
2936 3) The sum of all the numbers in the list.

```
2937 {73,12,80,37,56,94,40,21,7,24,15,14,82,93,32,74,22,68,65,52,85,61,46,86,25}
```

```
2938 Hint: the following program finds the largest number in a list and it can
2939 be used as a starting point for solving this exercise.
```

```
2940 %mathpiper
```

```
2941 /*
2942  The variable that keeps track of the largest number encountered so
2943  far needs to be initialized to the lowest possible value it may
2944  hold. Why?
2945 */
```

```
2946 largest := 0;
```

```
2947 numbersList := {4,6,7,9,2,1,3};
```

```
2948 index := 1;
```

```
2949 While(index <= Length(numbersList) )
2950 [
2951     Echo("Largest: ", largest);
2952     If(numbersList[index] > largest, largest := numbersList[index]);
2953     index := index + 1;
2954 ];
```

```
2957 Echo("The largest number in the list is: ", largest);
```

```
2958 %/mathpiper
```

2959 **15.4 The ForEach() Looping Function**

```
2960 The ForEach() function uses a loop to index through a list like the While()
2961 function does, but it is more flexible and automatic. ForEach() also uses bodied
2962 notation like the While() function and here is its calling format:
```

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

```
2963 ForEach() selects each expression in a list in turn, assigns it to the passed-in
2964 "variable", and then executes the expressions that are inside of "body".
2965 Therefore, body is executed once for each expression in the list.
```

2966 **15.5 Print All The Values In A List Using A ForEach() function**

```
2967 This example shows how ForEach() can be used to print all of the items in a list:
```

```
2968 %mathpiper
2969 //Print all values in a list.
2970 ForEach(value, {50,51,52,53,54,55,56,57,58,59})
2971 [
2972     Echo(value);
2973 ];
2974 %/mathpiper
2975 %output,preserve="false"
2976 Result: True
2977
2978 Side Effects:
2979 50
2980 51
2981 52
2982 53
2983 54
2984 55
2985 56
2986 57
2987 58
2988 59
2989 . %/output
```

2990 **15.6 Calculate The Sum Of The Numbers In A List Using ForEach()**

2991 In previous examples, counting code in the form **x := x + 1** was used to count
2992 how many times a While loop was executed. The following program uses a
2993 **ForEach()** function and a line of code similar to this counter to calculate the
2994 **sum of the numbers in a list:**

```
2995 %mathpiper
2996 /*
2997     This program calculates the sum of the numbers
2998     in a list.
2999 */
3000 //This variable is used to accumulate the sum.
3001 sum := 0;
3002 ForEach(number, {1,2,3,4,5,6,7,8,9,10} )
3003 [
3004     /*
3005         Add the contents of x to the contents of sum
```

```
3006         and place the result back into sum.
3007     */
3008     sum := sum + number;
3009
3010     //Print the sum as it is being accumulated.
3011     Write(sum,,);
3012 ];
3013 NewLine(); NewLine();
3014 Echo("The sum of the numbers in the list = ", sum);
3015 %/mathpiper
3016     %output,preserve="false"
3017     Result: True
3018
3019     Side Effects:
3020     1,3,6,10,15,21,28,36,45,55,
3021
3022     The sum of the numbers in the list = 55
3023 . %/output
```

3024 In the above program, the integers **1** through **10** were manually placed into a list
3025 by typing them individually. This method is limited because only a relatively
3026 small number of integers can be placed into a list this way. The following section
3027 discusses an operator which can be used to automatically place a large number
3028 of integers into a list with very little typing.

3029 15.7 The .. Range Operator

```
first .. last
```

3030 A programmer often needs to create a list which contains **consecutive integers**
3031 and the **.. "range"** operator can be used to do this. The **first** integer in the list is
3032 placed before the **..** operator and the **last** integer in the list is placed after it
3033 (**Note: there must be a space immediately to the left of the .. operator**
3034 **and a space immediately to the right of it or an error will be generated.**).
3035 Here are some examples:

```
3036 In> 1 .. 10
3037 Result> {1,2,3,4,5,6,7,8,9,10}
3038 In> 10 .. 1
3039 Result> {10,9,8,7,6,5,4,3,2,1}
3040 In> 1 .. 100
3041 Result> {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
```



```
3042      21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,  
3043      38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,  
3044      55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,  
3045      72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,  
3046      89,90,91,92,93,94,95,96,97,98,99,100}
```

```
3047 In> -10 .. 10  
3048 Result> {-10,-9,-8,-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,8,9,10}
```

3049 As these examples show, the `..` operator can generate lists of integers in
3050 ascending order and descending order. It can also generate lists that are very
3051 large and ones that contain negative integers.

3052 Remember, though, if one or both of the spaces around the `..` are omitted, an
3053 error is generated:

```
3054 In> 1..3  
3055 Result>  
3056 Error parsing expression, near token .3.
```

3057 **15.8 Using ForEach() With The Range Operator To Print The Prime** 3058 **Numbers Between 1 And 100**

3059 The following program shows how to use a **ForEach()** function instead of a
3060 **While()** function to print the prime numbers between 1 and 100. Notice that
3061 loops that are implemented with **ForEach()** often require less typing than
3062 their **While()** based equivalents:

```
3063 %mathpiper  
3064 /*  
3065   This program prints the prime integers between 1 and 100 using  
3066   a ForEach() function instead of a While() function. Notice that  
3067   the ForEach() version requires less typing than the While()  
3068   version.  
3069 */  
3070 ForEach(number, 1 .. 100)  
3071 [  
3072   If(IsPrime(number), Write(number,,) );  
3073 ];  
3074 %/mathpiper  
3075 %output,preserve="false"  
3076 Result: True  
3077  
3078 Side Effects:  
3079 2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,  
3080 73,79,83,89,97,
```

```
3081 .    %/output
```

3082 15.8.1 Using ForEach() And The Range Operator To Place The Prime 3083 Numbers Between 1 And 50 Into A List

3084 A ForEach() function can also be used to place values in a list, just the the
3085 While() function can:

```
3086 %mathpiper
```

```
3087 /*  
3088     Place the prime numbers between 1 and 50 into  
3089     a list using a ForEach() function.  
3090 */  
  
3091 //Create a new list.  
3092 primesList := {};  
  
3093 ForEach(number, 1 .. 50)  
3094 [  
3095     /*  
3096         If number is prime, append it to the end of the list and  
3097         then assign the new list that is created to the variable  
3098         'primes'.  
3099     */  
3100     If(IsPrime(number), primesList := Append(primesList, number ) );  
3101 ];
```

```
3102 //Print information about the primes that were found.  
3103 Echo("Primes ", primesList);  
3104 Echo("The number of primes in the list = ", Length(primesList) );  
3105 Echo("The first number in the list = ", primesList[1] );
```

```
3106 %/mathpiper
```

```
3107     %output,preserve="false"  
3108     Result: True  
3109  
3110     Side Effects:  
3111     Primes {2,3,5,7,11,13,17,19,23,29,31,37,41,43,47}  
3112     The number of primes in the list = 15  
3113     The first number in the list = 2  
3114 .    %/output
```

3115 As can be seen from the above examples, the **ForEach()** function and the **range**
3116 **operator** can do a significant amount of work with very little typing. You will
3117 discover in the next section that MathPiper has functions which are even more
3118 powerful than these two.

3119 15.8.2 Exercises

3120 For the following exercises, create a new MathRider worksheet file called
3121 **book_1_section_15b_exercises_<your first name>_<your last name>.mrw.**
3122 **(Note: there are no spaces in this file name).** For example, John Smith's
3123 worksheet would be called:

3124 **book_1_section_15b_exercises_john_smith.mrw.**

3125 After this worksheet has been created, place your answer for each exercise that
3126 requires a fold into its own fold in this worksheet. Place a title attribute in the
3127 start tag of each fold which indicates the exercise the fold contains the solution
3128 to. The folds you create should look similar to this one:

3129 `%mathpiper,title="Exercise 1"`

3130 `//Sample fold.`

3131 `%/mathpiper`

3132 If an exercise uses the MathPiper console instead of a fold, copy the work you
3133 did in the console into the worksheet so it can be saved but do not put it in a fold.

3134 15.8.3 Exercise 1

3135 Create a program that uses a **ForEach()** function and an **IsOdd()** predicate
3136 function to analyze the following list and then print the number of odd
3137 numbers it contains.

3138 `{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}`

3139 15.8.4 Exercise 2

3140 Create a program that uses a **ForEach()** function and an **IsNegativeNumber()**
3141 function to copy all of the negative numbers in the following list into a
3142 new list. Use the variable **negativeNumbersList** to hold the new list.
3143 Print the contents of the list after it has been created.

3144 `{36,-29,-33,-6,14,7,-16,-3,-14,37,-38,-8,-45,-21,-26,6,6,38,-20,33,41,-`
3145 `4,24,37,40,29}`

3146 15.8.5 Exercise 3

3147 Create one program that uses a single **ForEach()** function to analyze the
3148 following list and then print the following information about it:

- 3149 1) The largest number in the list.
3150 2) The smallest number in the list.
3151 3) The sum of all the numbers in the list.

3152 {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}

3153 **15.8.6 Exercise 4**

3154 Create one program that does the following: 1) uses a **While loop** to make a
3155 list that contains **1000 random integers** between **1** and **100** inclusive. 2)
3156 uses a **ForEach()** function to determine how many integers in the list are
3157 **prime** and use an **Echo()** function to print this total.

3158 **16 Functions & Operators Which Loop Internally**

3159 Looping is such a useful capability that MathPiper has many functions which
3160 loop internally. Now that you have some experience with loops, you can use this
3161 experience to help you imagine how these functions use loops to process the
3162 information that is passed to them.

3163 **16.1 Functions & Operators Which Loop Internally To Process Lists**

3164 This section discusses a number of functions that use loops to process lists.

3165 **16.1.1 TableForm()**

```
TableForm(list)
```

3166 The **TableForm()** function prints the contents of a list in the form of a table.
3167 Each member in the list is printed on its own line and this sometimes makes the
3168 contents of the list easier to read:

```
3169 In> testList := {2,4,6,8,10,12,14,16,18,20}  
3170 Result> {2,4,6,8,10,12,14,16,18,20}
```

```
3171 In> TableForm(testList)  
3172 Result> True  
3173 Side Effects>  
3174 2  
3175 4  
3176 6  
3177 8  
3178 10  
3179 12  
3180 14  
3181 16  
3182 18  
3183 20
```

3184 **16.1.2 Contains()**

3185 The **Contains()** function searches a list to determine if it contains a given
3186 expression. If it finds the expression, it returns **True** and if it doesn't find the
3187 expression, it returns **False**. Here is the calling format for Contains():

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

3188 The following code shows Contains() being used to locate a number in a list:

```
3189 In> Contains({50,51,52,53,54,55,56,57,58,59}, 53)
3190 Result> True
```

```
3191 In> Contains({50,51,52,53,54,55,56,57,58,59}, 75)
3192 Result> False
```

3193 The **Not()** function can also be used with predicate functions like Contains() to
3194 change their results to the opposite truth value:

```
3195 In> Not Contains({50,51,52,53,54,55,56,57,58,59}, 75)
3196 Result> True
```

3197 16.1.3 Find()

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

3198 The **Find()** function searches a list for the first occurrence of a given expression.
3199 If the expression is found, the **position of its first occurrence** is returned and
3200 if it is not found, **-1** is returned:

```
3201 In> Find({23, 15, 67, 98, 64}, 15)
3202 Result> 2
```

```
3203 In> Find({23, 15, 67, 98, 64}, 8)
3204 Result> -1
```

3205 16.1.4 Count()

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

3206 **Count()** determines the number of times a given expression occurs in a list:

```
3207 In> testList := {a,b,b,c,c,c,d,d,d,d,e,e,e,e,e}
3208 Result> {a,b,b,c,c,c,d,d,d,d,e,e,e,e,e}
```

```
3209 In> Count(testList, c)
3210 Result> 3
```

```
3211 In> Count(testList, e)
3212 Result> 5
```

```
3213 In> Count(testList, z)
3214 Result> 0
```

3215 **16.1.5 Select()**

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

3216 **Select()** returns a list that contains all the expressions in a list which make a
3217 given predicate function return **True**:

```
3218 In> Select("IsPositiveInteger", {46,87,59,-27,11,86,-21,-58,-86,-52})  
3219 Result> {46,87,59,11,86}
```

3220 In this example, notice that the **name** of the predicate function is passed to
3221 **Select()** in **double quotes**. There are other ways to pass a predicate function to
3222 **Select()** but these are covered in a later section.

3223 Here are some further examples which use the **Select()** function:

```
3224 In> Select("IsOdd", {16,14,82,92,33,74,99,67,65,52})  
3225 Result> {33,99,67,65}
```

```
3226 In> Select("IsEven", {16,14,82,92,33,74,99,67,65,52})  
3227 Result> {16,14,82,92,74,52}
```

```
3228 In> Select("IsPrime", 1 .. 75)  
3229 Result> {2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73}
```

3230 Notice how the third example uses the **..** operator to automatically generate a list
3231 of consecutive integers from 1 to 75 for the **Select()** function to analyze.

3232 **16.1.6 The Nth() Function & The [] Operator**

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

3233 The **Nth()** function simply returns the expression which is at a given position in
3234 a list. This example shows the **third** expression in a list being obtained:

```
3235 In> testList := {a,b,c,d,e,f,g}  
3236 Result> {a,b,c,d,e,f,g}
```

```
3237 In> Nth(testList, 3)  
3238 Result> c
```

3239 As discussed earlier, the **[]** operator can also be used to obtain a single
3240 expression from a list:

```
3241 In> testList[3]
3242 Result> c
```

3243 The **[]** operator can even obtain a single expression directly from a list without
3244 needing to use a variable:

```
3245 In> {a,b,c,d,e,f,g}[3]
3246 Result> c
```

3247 **16.1.7 The : Prepend Operator**

```
expression : list
```

3248 The prepend operator is a colon **:** and it can be used to add an expression to the
3249 beginning of a list:

```
3250 In> testList := {b,c,d}
3251 Result> {b,c,d}

3252 In> testList := a:testList
3253 Result> {a,b,c,d}
```

3254 **16.1.8 Concat()**

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

3255 The Concat() function is short for "concatenate" which means to join together
3256 sequentially. It takes two or more lists and joins them together into a single
3257 larger list:

```
3258 In> Concat({a,b,c}, {1,2,3}, {x,y,z})
3259 Result> {a,b,c,1,2,3,x,y,z}
```

3260 **16.1.9 Insert(), Delete(), & Replace()**

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

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

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


3261 **Insert()** inserts an expression into a list at a given index, **Delete()** deletes an
3262 expression from a list at a given index, and **Replace()** replaces an expression in
3263 a list at a given index with another expression:

```
3264 In> testList := {a,b,c,d,e,f,g}
3265 Result> {a,b,c,d,e,f,g}

3266 In> testList := Insert(testList, 4, 123)
3267 Result> {a,b,c,123,d,e,f,g}

3268 In> testList := Delete(testList, 4)
3269 Result> {a,b,c,d,e,f,g}

3270 In> testList := Replace(testList, 4, xxx)
3271 Result> {a,b,c,xxx,e,f,g}
```

3272 16.1.10 Take()

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

3273 **Take()** obtains a sublist from the **beginning** of a list, the **end** of a list, or the
3274 **middle** of a list. The expressions in the list that are not taken are discarded.

3275 A **positive** integer passed to Take() indicates how many expressions should be
3276 taken from the **beginning** of a list:

```
3277 In> testList := {a,b,c,d,e,f,g}
3278 Result> {a,b,c,d,e,f,g}

3279 In> Take(testList, 3)
3280 Result> {a,b,c}
```

3281 A **negative** integer passed to Take() indicates how many expressions should be
3282 taken from the **end** of a list:

```
3283 In> Take(testList, -3)
3284 Result> {e,f,g}
```

3285 Finally, if a **two member list** is passed to Take() it indicates the **range** of
3286 expressions that should be taken from the **middle** of a list. The **first** value in the
3287 passed-in list specifies the **beginning** index of the range and the **second** value
3288 specifies its **end**:

```
3289 In> Take(testList, {3,5})
3290 Result> {c,d,e}
```

3291 **16.1.11 Drop()**

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

3292 **Drop()** does the opposite of Take() in that it **drops** expressions from the
3293 **beginning** of a list, the **end** of a list, or the **middle** of a list and **returns a list**
3294 **which contains the remaining expressions.**

3295 A **positive** integer passed to Drop() indicates how many expressions should be
3296 dropped from the **beginning** of a list:

```
3297 In> testList := {a,b,c,d,e,f,g}
3298 Result> {a,b,c,d,e,f,g}
```

```
3299 In> Drop(testList, 3)
3300 Result> {d,e,f,g}
```

3301 A **negative** integer passed to Drop() indicates how many expressions should be
3302 dropped from the **end** of a list:

```
3303 In> Drop(testList, -3)
3304 Result> {a,b,c,d}
```

3305 Finally, if a **two member list** is passed to Drop() it indicates the **range** of
3306 expressions that should be dropped from the **middle** of a list. The **first** value in
3307 the passed-in list specifies the **beginning** index of the range and the **second**
3308 value specifies its **end**:

```
3309 In> Drop(testList, {3,5})
3310 Result> {a,b,f,g}
```

3311 **16.1.12 FillList()**

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

3312 The FillList() function simply creates a list which is of size "length" and fills it
3313 with "length" copies of the given expression:

```
3314 In> FillList(a, 5)
3315 Result> {a,a,a,a,a}
```

```
3316 In> FillList(42,8)
3317 Result> {42,42,42,42,42,42,42,42}
```

3318 **16.1.13 RemoveDuplicates()**

```
RemoveDuplicates(list)
```

3319 **RemoveDuplicates()** removes any duplicate expressions that are contained in a
3320 list:

```
3321 In> testList := {a,a,b,c,c,b,b,a,b,c,c}
```

```
3322 Result> {a,a,b,c,c,b,b,a,b,c,c}
```

```
3323 In> RemoveDuplicates(testList)
```

```
3324 Result> {a,b,c}
```

3325 **16.1.14 Reverse()**

```
Reverse(list)
```

3326 **Reverse()** reverses the order of the expressions in a list:

```
3327 In> testList := {a,b,c,d,e,f,g,h}
```

```
3328 Result> {a,b,c,d,e,f,g,h}
```

```
3329 In> Reverse(testList)
```

```
3330 Result> {h,g,f,e,d,c,b,a}
```

3331 **16.1.15 Partition()**

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

3332 The **Partition()** function breaks a list into sublists of size "partition_size":

```
3333 In> testList := {a,b,c,d,e,f,g,h}
```

```
3334 Result> {a,b,c,d,e,f,g,h}
```

```
3335 In> Partition(testList, 2)
```

```
3336 Result> {{a,b},{c,d},{e,f},{g,h}}
```

3337 If the partition_size does not divide the length of the list **evenly**, the remaining
3338 elements are discarded:

```
3339 In> Partition(testList, 3)
```

```
3340 Result> {{h,b,c},{d,e,f}}
```

3341 The number of elements that Partition() will discard can be calculated by
3342 dividing the length of a list by the partition size and obtaining the **remainder**:

```
3343 In> Length(testList) % 3  
3344 Result> 2
```

3345 Remember that % is the remainder operator. It divides two integers and returns
3346 their remainder.

3347 **16.1.16 Table()**

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

3348 The Table() function creates a list of values by doing the following:

- 3349 1) Generating a sequence of values between a "begin_value" and an
3350 "end_value" with each value being incremented by the "step_amount".
- 3351 2) Placing each value in the sequence into the specified "variable", one value
3352 at a time.
- 3353 3) Evaluating the defined "expression" (which contains the defined "variable")
3354 for each value, one at a time.
- 3355 4) Placing the result of each "expression" evaluation into the result list.

3356 This example generates a list which contains the integers 1 through 10:

```
3357 In> Table(x, x, 1, 10, 1)  
3358 Result> {1,2,3,4,5,6,7,8,9,10}
```

3359 Notice that the expression in this example is simply the variable 'x' itself with no
3360 other operations performed on it.

3361 The following example is similar to the previous one except that its expression
3362 multiplies 'x' by 2:

```
3363 In> Table(x*2, x, 1, 10, 1)  
3364 Result> {2,4,6,8,10,12,14,16,18,20}
```

3365 Lists which contain decimal values can also be created by setting the
3366 "step_amount" to a decimal:

```
3367 In> Table(x, x, 0, 1, .1)  
3368 Result> {0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1}
```

3369 **16.1.17 HeapSort()**

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

3370 **HeapSort()** sorts the elements of **list** into the order indicated by **compare** with
3371 **compare** typically being the **less than** operator "<" or the **greater than**
3372 operator ">":

```
3373 In> HeapSort({4,7,23,53,-2,1}, "<");  
3374 Result: {-2,1,4,7,23,53}
```

```
3375 In> HeapSort({4,7,23,53,-2,1}, ">");  
3376 Result: {53,23,7,4,1,-2}
```

```
3377 In> HeapSort({1/2,3/5,7/8,5/16,3/32}, "<")  
3378 Result: {3/32,5/16,1/2,3/5,7/8}
```

```
3379 In> HeapSort({.5,3/5,.76,5/16,3/32}, "<")  
3380 Result: {3/32,5/16,.5,3/5,.76}
```

3381 **16.2 Functions That Work With Integers**

3382 This section discusses various functions which work with integers. Some of
3383 these functions also work with non-integer values and their use with non-
3384 integers is discussed in other sections.

3385 **16.2.1 RandomIntegerVector()**

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

3386 A vector is a list that does not contain other lists. **RandomIntegerVector()**
3387 creates a list of size "length" that contains random integers that are no lower
3388 than "lowest_possible" and no higher than "highest possible". The following
3389 example creates **10** random integers between **1** and **99** inclusive:

```
3390 In> RandomIntegerVector(10, 1, 99)  
3391 Result> {73,93,80,37,55,93,40,21,7,24}
```

3392 **16.2.2 Max() & Min()**

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

3393 If two values are passed to **Max()**, it determines which one is larger:

```
3394 In> Max(10, 20)
```

3395 `Result> 20`

3396 If a list of values are passed to `Max()`, it finds the largest value in the list:

3397 `In> testList := RandomIntegerVector(10, 1, 99)`

3398 `Result> {73,93,80,37,55,93,40,21,7,24}`

3399 `In> Max(testList)`

3400 `Result> 93`

3401 The **Min()** function is the opposite of the `Max()` function.

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

```
Min(list)
```

3402 If two values are passed to `Min()`, it determines which one is smaller:

3403 `In> Min(10, 20)`

3404 `Result> 10`

3405 If a list of values are passed to `Min()`, it finds the smallest value in the list:

3406 `In> testList := RandomIntegerVector(10, 1, 99)`

3407 `Result> {73,93,80,37,55,93,40,21,7,24}`

3408 `In> Min(testList)`

3409 `Result> 7`

3410 **16.2.3 Div() & Mod()**

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

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

3411 **Div()** stands for "divide" and determines the whole number of times a divisor
3412 goes into a dividend:

3413 `In> Div(7, 3)`

3414 `Result> 2`

3415 **Mod()** stands for "modulo" and it determines the remainder that results when a
3416 dividend is divided by a divisor:

3417 `In> Mod(7,3)`

3418 `Result> 1`

3419 The remainder/modulo operator **%** can also be used to calculate a remainder:

```
3420 In> 7 % 2
3421 Result> 1
```

3422 **16.2.4 Gcd()**

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

3423 GCD stands for Greatest Common Divisor and the **Gcd()** function determines the
3424 greatest common divisor of the values that are passed to it.

3425 If two integers are passed to Gcd(), it calculates their greatest common divisor:

```
3426 In> Gcd(21, 56)
3427 Result> 7
```

3428 If a list of integers are passed to Gcd(), it finds the greatest common divisor of all
3429 the integers in the list:

```
3430 In> Gcd({9, 66, 123})
3431 Result> 3
```

3432 **16.2.5 Lcm()**

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

3433 LCM stands for Least Common Multiple and the **Lcm()** function determines the
3434 least common multiple of the values that are passed to it.

3435 If two integers are passed to Lcm(), it calculates their least common multiple:

```
3436 In> Lcm(14, 8)
3437 Result> 56
```

3438 If a list of integers are passed to Lcm(), it finds the least common multiple of all
3439 the integers in the list:

```
3440 In> Lcm({3, 7, 9, 11})
3441 Result> 693
```

3442 **16.2.6 Sum()**

```
Sum(list)
```

3443 **Sum()** can find the sum of a list that is passed to it:

3444 In> testList := RandomIntegerVector(10,1,99)

3445 Result> {73,93,80,37,55,93,40,21,7,24}

3446 In> Sum(testList)

3447 Result> 523

3448 In> testList := 1 .. 10

3449 Result> {1,2,3,4,5,6,7,8,9,10}

3450 In> Sum(testList)

3451 Result> 55

3452 **16.2.7 Product()**

```
Product(list)
```

3453 This function has two calling formats, only one of which is discussed here.

3454 **Product(list)** multiplies all the expressions in a list together and returns their
3455 product:

3456 In> Product({1,2,3})

3457 Result> 6

3458 **16.3 Exercises**3459 For the following exercises, create a new MathRider worksheet file called
3460 **book_1_section_16_exercises_<your first name>_<your last name>.mrw.**
3461 **(Note: there are no spaces in this file name).** For example, John Smith's
3462 worksheet would be called:3463 **book_1_section_16_exercises_john_smith.mrw.**3464 After this worksheet has been created, place your answer for each exercise that
3465 requires a fold into its own fold in this worksheet. Place a title attribute in the
3466 start tag of each fold which indicates the exercise the fold contains the solution
3467 to. The folds you create should look similar to this one:

3468 %mathpiper,title="Exercise 1"

3469 //Sample fold.

3470 [%/mathpiper](#)

3471 If an exercise uses the MathPiper console instead of a fold, copy the work you
3472 did in the console into the worksheet so it can be saved but do not put it in a fold.

3473 16.3.1 Exercise 1

3474 Create a program that uses **RandomIntegerVector()** to create a 100 member
3475 list that contains random integers between 1 and 5 inclusive. Use **Count()**
3476 to determine how many of each digit 1-5 are in the list and then print this
3477 information. Hint: you can use the **HeapSort()** function to sort the
3478 generated list to make it easier to check if your program is counting
3479 correctly.

3480 16.3.2 Exercise 2

3481 Create a program that uses **RandomIntegerVector()** to create a 100 member
3482 list that contains random integers between 1 and 50 inclusive and use
3483 **Contains()** to determine if the number 25 is in the list. Print "25 was in
3484 the list." if 25 was found in the list and "25 was not in the list." if it
3485 wasn't found.

3486 16.3.3 Exercise 3

3487 Create a program that uses **RandomIntegerVector()** to create a 100 member
3488 list that contains random integers between 1 and 50 inclusive and use
3489 **Find()** to determine if the number 10 is in the list. Print the position of
3490 10 if it was found in the list and "10 was not in the list." if it wasn't
3491 found.

3492 16.3.4 Exercise 4

3493 Create a program that uses **RandomIntegerVector()** to create a 100 member
3494 list that contains random integers between 0 and 3 inclusive. Use **Select()**
3495 with the **IsNonZeroInteger()** predicate function to obtain all of the nonzero
3496 integers in this list.

3497 16.3.5 Exercise 5

3498 Create a program that uses **Table()** to obtain a list which contains the
3499 squares of the integers between 1 and 10 inclusive.

3500 17 Nested Loops

3501 Now that you have seen how to solve problems with single loops, it is time to
3502 discuss what can be done when a loop is placed inside of another loop. A loop
3503 that is placed **inside** of another loop it is called a **nested loop** and this nesting
3504 can be extended to numerous levels if needed. This means that loop 1 can have
3505 loop 2 placed inside of it, loop 2 can have loop 3 placed inside of it, loop 3 can
3506 have loop 4 placed inside of it, and so on.

3507 Nesting loops allows the programmer to accomplish an enormous amount of
3508 work with very little typing.

3509 17.1 Generate All The Combinations That Can Be Entered Into A Two Digit 3510 Wheel Lock Using A Nested Loop



3511 The following program generates all the combinations that can be entered into a
3512 two digit wheel lock. It uses a nested loop to accomplish this with the "**inside**"
3513 nested loop being used to generate **one's place** digits and the "**outside**" loop
3514 being used to generate **ten's place** digits.

```
3515 %mathpiper
3516 /*
3517    Generate all the combinations can be entered into a two
3518    digit wheel lock.
3519 */
3520 combinationsList := {};
3521 ForEach(digit1, 0 .. 9) //This loop is called the "outside" loop.
```

```
3522 [
3523     ForEach(digit2, 0 .. 9)//This loop is called the "inside" loop.
3524     [
3525         combinationsList := Append(combinationsList, {digit1, digit2});
3526     ];
3527 ];

3528 Echo(TableForm(combinationsList));

3529 %/mathpiper

3530     %output,preserve="false"
3531     Result: True
3532
3533     Side Effects:
3534     {0,0}
3535     {0,1}
3536     {0,2}
3537     {0,3}
3538     {0,4}
3539     {0,5}
3540     {0,6}
3541     .
3542     . //The middle of the list has not been shown.
3543     .
3544     {9,3}
3545     {9,4}
3546     {9,5}
3547     {9,6}
3548     {9,7}
3549     {9,8}
3550     {9,9}
3551     True
3552 . %/output
```

3553 The relationship between the outside loop and the inside loop is interesting
3554 because each time the **outside loop cycles once**, the **inside loop cycles 10**
3555 **times**. Study this program carefully because nested loops can be used to solve a
3556 wide range of problems and therefore understanding how they work is
3557 important.

3558 17.2 Exercises

3559 For the following exercises, create a new MathRider worksheet file called
3560 **book_1_section_17_exercises_<your first name>_<your last name>.mrw**.
3561 (**Note: there are no spaces in this file name**). For example, John Smith's
3562 worksheet would be called:
3563 **book_1_section_17_exercises_john_smith.mrw**.

3564 After this worksheet has been created, place your answer for each exercise that
3565 requires a fold into its own fold in this worksheet. Place a title attribute in the
3566 start tag of each fold which indicates the exercise the fold contains the solution
3567 to. The folds you create should look similar to this one:

3568 `%mathpiper,title="Exercise 1"`

3569 `//Sample fold.`

3570 `%/mathpiper`

3571 If an exercise uses the MathPiper console instead of a fold, copy the work you
3572 did in the console into the worksheet so it can be saved but do not put it in a fold.

3573 **17.2.1 Exercise 1**

3574 Create a program that will generate all of the combinations that can be
3575 entered into a three digit wheel lock. (Hint: a triple nested loop can be
3576 used to accomplish this.)

3577 18 User Defined Functions

3578 In computer programming, a **function** is a named section of code that can be
3579 **called** from other sections of code. **Values** can be sent to a function for
3580 processing as part of the **call** and a function always returns a value as its result.
3581 A function can also generate side effects when it is called and side effects have
3582 been covered in earlier sections.

3583 The values that are sent to a function when it is called are called **arguments** or
3584 **actual parameters** and a function can accept 0 or more of them. These
3585 arguments are placed within parentheses.

3586 MathPiper has many predefined functions (some of which have been discussed in
3587 previous sections) but users can create their own functions too. The following
3588 program creates a function called **addNums()** which takes two numbers as
3589 arguments, adds them together, and returns their sum back to the calling code
3590 as a result:

```
3591 In> addNums(num1,num2) := num1 + num2
3592 Result> True
```

3593 This line of code defined a new function called **addNums** and specified that it
3594 will accept two values when it is called. The **first** value will be placed into the
3595 variable **num1** and the **second** value will be placed into the variable **num2**.

3596 Variables like num1 and num2 which are used in a function to accept values from
3597 calling code are called **formal parameters**. **Formal parameter variables** are
3598 used inside a function to process the **values/actual parameters/arguments**
3599 that were placed into them by the calling code.

3600 The code on the **right side** of the **assignment operator** is **bound** to the
3601 function name "**addNums**" and it is executed each time **addNums()** is called.
3602 The following example shows the new **addNums()** function being called multiple
3603 times with different values being passed to it:

```
3604 In> addNums(2,3)
3605 Result> 5
```

```
3606 In> addNums(4,5)
3607 Result> 9
```

```
3608 In> addNums(9,1)
3609 Result> 10
```

3610 Notice that, unlike the functions that come with MathPiper, we chose to have this
3611 function's name start with a **lower case letter**. We could have had addNums()
3612 begin with an upper case letter but it is a **convention** in MathPiper for **user**

3613 **defined function names to begin with a lower case letter to distinguish**
3614 **them from the functions that come with MathPiper.**

3615 The values that are returned from user defined functions can also be assigned to
3616 variables. The following example uses a %mathpiper fold to define a function
3617 called **evenIntegers()** and then this function is used in the MathPiper console:

```
3618 %mathpiper
3619 evenIntegers(endInteger) :=
3620 [
3621     resultList := {};
3622     x := 2;
3623     While(x <= endInteger)
3624     [
3625         resultList := Append(resultList, x);
3626         x := x + 2;
3627     ];
3628 ];
3629 /*
3630 The result of the last expression which is executed in a function
3631 is the result that the function returns to the caller. In this case,
3632 resultList is purposely being executed last so that its contents are
3633 returned to the caller.
3634 */
3635 resultList;
3636 ];
3637
3638 %/mathpiper
3639 %output,preserve="false"
3640 Result: True
3641 . %/output
3642 In> a := evenIntegers(10)
3643 Result> {2,4,6,8,10}
3644 In> Length(a)
3645 Result> 5
```

3646 The function **evenIntegers()** returns a list which contains all the even integers
3647 from 2 up through the value that was passed into it. The fold was first executed
3648 in order to define the **evenIntegers()** function and make it ready for use. The
3649 **evenIntegers()** function was then called from the MathPiper console and 10
3650 was passed to it.

3651 After the function was finished executing, it returned a list of even integers as a

3652 result and this result was assigned to the variable 'a'. We then passed the list
3653 that was assigned to 'a' to the **Length()** function in order to determine its size.

3654 **18.1 Global Variables, Local Variables, & Local()**

3655 The new **evenIntegers()** function seems to work well, but there is a problem.
3656 The variables 'x' and **resultList** were defined inside the function as **global**
3657 **variables** which means they are accessible from anywhere, including from
3658 within other functions, within other folds (as shown here):

```
3659 %mathpiper
3660 Echo(x, ",", resultList);
3661 %/mathpiper
3662     %output,preserve="false"
3663     Result: True
3664
3665     Side Effects:
3666     12 , {2,4,6,8,10}
3667 .    %/output
```

3668 and from within the MathPiper console:

```
3669 In> x
3670 Result> 12
3671 In> resultList
3672 Result> {2,4,6,8,10}
```

3673 **Using global variables inside of functions is usually not a good idea**
3674 because code in other functions and folds might already be using (or will use) the
3675 same variable names. Global variables which have the same name are the same
3676 variable. When one section of code changes the value of a given global variable,
3677 the value is changed everywhere that variable is used and this will eventually
3678 cause problems.

3679 In order to prevent errors being caused by global variables having the same
3680 name, a function named **Local()** can be called inside of a function to define what
3681 are called **local variables**. A **local variable** is only accessible inside the
3682 function it has been defined in, even if it has the same name as a global variable.
3683 The following example shows a second version of the **evenIntegers()** function
3684 which uses **Local()** to make 'x' and **resultList** local variables:

```
3685 %mathpiper
3686 /*
3687  This version of evenIntegers() uses Local() to make
3688  x and resultList local variables
3689 */
3690 evenIntegers(endInteger) :=
3691 [
3692     Local(x,resultList);
3693     resultList := {};
3694
3695     x := 2;
3696
3697     While(x <= endInteger)
3698     [
3699         resultList := Append(resultList, x);
3700
3701         x := x + 2;
3702     ];
3703     /*
3704     The result of the last expression which is executed in a function
3705     is the result that the function returns to the caller. In this case,
3706     resultList is purposely being executed last so that its contents are
3707     returned to the caller.
3708     */
3709     resultList;
3710 ];
3711 %/mathpiper
3712     %output,preserve="false"
3713     Result: True
3714 .    %/output
```

3715 We can verify that '**x**' and **resultList** are now local variables by first clearing
3716 them, calling **evenIntegers()**, and then seeing what '**x**' and **resultList** contain:

```
3717 In> Clear(x, resultList)
3718 Result> True
3719 In> evenIntegers(10)
3720 Result> {2,4,6,8,10}
3721 In> x
3722 Result> x
3723 In> resultList
3724 Result> resultList
```


3725 18.2 Exercises

3726 For the following exercises, create a new MathRider worksheet file called
3727 **book_1_section_18_exercises_<your first name>_<your last name>.mrw.**
3728 **(Note: there are no spaces in this file name).** For example, John Smith's
3729 worksheet would be called:

3730 **book_1_section_18_exercises_john_smith.mrw.**

3731 After this worksheet has been created, place your answer for each exercise that
3732 requires a fold into its own fold in this worksheet. Place a title attribute in the
3733 start tag of each fold which indicates the exercise the fold contains the solution
3734 to. The folds you create should look similar to this one:

```
3735 %mathpiper,title="Exercise 1"
```

```
3736 //Sample fold.
```

```
3737 %/mathpiper
```

3738 If an exercise uses the MathPiper console instead of a fold, copy the work you
3739 did in the console into the worksheet so it can be saved but do not put it in a fold.

3740 18.2.1 Exercise 1

3741 Create a function called **tenOddIntegers()** which returns a list which
3742 contains 10 random odd integers between 1 and 99 inclusive.

3743 18.2.2 Exercise 2

3744 Create a function called **convertStringToList(string)** which takes a string
3745 as a parameter and returns a list which contains all of the characters in
3746 the string. Here is an example of how the function should work:

```
3747 In> convertStringToList("Hello friend!")  
3748 Result> {"H","e","l","l","o"," ","f","r","i","e","n","d","!"}
```

```
3749 In> convertStringToList("Computer Algebra System")  
3750 Result> {"C","o","m","p","u","t","e","r"," ","A","l","g","e","b","r","a","  
3751 ","S","y","s","t","e","m"}
```

3752 19 Miscellaneous topics

3753 19.1 Incrementing And Decrementing Variables With The ++ And -- 3754 Operators

3755 Up until this point we have been adding 1 to a variable with code in the form of **x**
3756 **:= x + 1** and subtracting 1 from a variable with code in the form of **x := x - 1**.
3757 Another name for **adding** 1 to a variable is **incrementing** it and **decrementing**
3758 a variable means to **subtract** 1 from it. Now that you have had some experience
3759 with these longer forms, it is time to show you shorter versions of them.

3760 19.1.1 Incrementing Variables With The ++ Operator

3761 The number 1 can be added to a variable by simply placing the ++ operator after
3762 it like this:

```
3763 In> x := 1  
3764 Result: 1
```

```
3765 In> x++;  
3766 Result: True
```

```
3767 In> x  
3768 Result: 2
```

3769 Here is a program that uses the ++ operator to increment a loop index variable:

```
3770 %mathpiper  
3771 index := 1;  
3772 While(index <= 10)  
3773 [  
3774     Echo(index);  
3775     index++; //The ++ operator increments the index variable.  
3776 ];  
3777  
3778 %/mathpiper  
3779 %output,preserve="false"  
3780 Result: True  
3781  
3782 Side Effects:  
3783 1  
3784 2
```

```
3785      3
3786      4
3787      5
3788      6
3789      7
3790      8
3791      9
3792     10
3793 .    %/output
```

3794 19.1.2 Decrementing Variables With The -- Operator

3795 The number 1 can be subtracted from a variable by simply placing the --
3796 operator after it like this:

```
3797 In> x := 1
3798 Result: 1

3799 In> x--;
3800 Result: True

3801 In> x
3802 Result: 0
```

3803 Here is a program that uses the -- operator to decrement a loop index variable:

```
3804 %mathpiper

3805 index := 10;

3806 While(index >= 1)
3807 [
3808     Echo(index);
3809     index--; //The -- operator decrements the index variable.
3810 ]

3812 %/mathpiper

3813 %output,preserve="false"
3814 Result: True
3815
3816 Side Effects:
3817 10
3818 9
3819 8
3820 7
3821 6
3822 5
```

```
3823         4
3824         3
3825         2
3826         1
3827     .    %/output
```

3828 19.1.3 The Break() Function

3829 The **Break()** function is used to end a loop early and here is its calling format:

```
Break()
```

3830 The following program has a While loop which is configured to loop 10 times.
3831 However, when the loop counter variable **index** reaches 5, the Break() function
3832 is called and this causes the loop to end early:

```
3833 %mathpiper
3834
3835 index := 1;
3836
3837 While(index <= 10)
3838 [
3839     Echo(index);
3840
3841     If(index = 5, Break());
3842
3843     index++;
3844 ];
3845
3846 %/mathpiper
3847
3848     %output,preserve="false"
3849     Result: True
3850
3851     Side Effects:
3852     1
3853     2
3854     3
3855     4
3856     5
3857 .    %/output
```

3857 When a Break() function is used to end a loop, it is called "**breaking out**" of the
3858 loop. Notice that only the numbers 1-5 are printed in this program.

3859 19.1.4 The Continue() Function

3860 The **Continue()** function is similar to the Break() function, except that instead of
3861 ending the loop, it simply causes it to **skip the remainder of the loop for the**
3862 **current loop iteration**. Here is the Continue() function's calling format:

```
Continue()
```

3863 The following program uses a While loop which is configured to print the
3864 integers from 0 to 8. However, the Continue() function is used to skip the
3865 execution of the Echo() function when the loop indexing variable **index** is equal
3866 to 5:

```
3867 %mathpiper
3868
3869 index := 0;
3870
3871 While(index < 8)
3872 [
3873     index++;
3874
3875     If(index = 5, Continue());
3876
3877     Echo(index);
3878 ];
3879
3880 %/mathpiper
3881
3882 %output,preserve="false"
3883 Result: True
3884
3885 Side Effects:
3886 1
3887 2
3888 3
3889 4
3890 6
3891 7
3892 8
3893 . %/output
```

3892 Notice that the number 5 is not printed when this program is executed.

3893 19.1.5 The Repeat() Function

3894 The **Repeat()** function is a looping function which is similar to While() and

3895 ForEach(), but it is simpler than these two. Here are the two calling formats for
3896 Repeat():

```
Repeat(count) body  
Repeat() body
```

3897 The first version of Repeat() simply takes an integer argument which indicates
3898 how many times it should loop. The following program shows how to use
3899 Repeat() to print 4 copies of the word "Hello":

```
3900 %mathpiper  
3901  
3902 Repeat(4)  
3903 [  
3904     Echo("Hello");  
3905 ];  
3906  
3907 %/mathpiper  
  
3908 %output,preserve="false"  
3909 Result: 4  
3910  
3911 Side Effects:  
3912 Hello  
3913 Hello  
3914 Hello  
3915 Hello  
3916 . %/output
```

3917 The second version of Repeat() does not take any arguments and it is designed to
3918 run as an **infinite loop**. The Break() function is then used to make the Repeat()
3919 function stop looping. The following program would print the loop indexing
3920 variable **index** forever, but the Break() function is used to stop the loop after **3**
3921 iterations:

```
3922 %mathpiper  
3923  
3924 index := 1;  
3925  
3926 loopCount := Repeat()  
3927 [  
3928     Echo(index);  
3929  
3930     If(index = 3, Break());  
3931
```

```
3932     index := index + 1;
3933 ];
3934
3935 Echo("Loop count: ", loopCount);
3936
3937 %/mathpiper
3938
3938     %output,preserve="false"
3939     Result: True
3940
3941     Side Effects:
3942     1
3943     2
3944     3
3945     Loop count: 2
3946 . %/output
```

3947 Notice that Repeat() returns the number of times it actually looped as a result
3948 and that this information is assigned to the variable **loopCount**.

3949 19.1.6 The EchoTime() Function

3950 Computers are extremely fast, but they still take time to execute programs.
3951 Sometimes it is important to determine how long it takes to evaluate a given
3952 expression in order to do things like determine if a section of code need to run
3953 quicker than it currently is or determine if one piece of code is slower than
3954 another. The EchoTime() function is a bodied function which is used to **time**
3955 **how long a section of code takes to run** and here is its calling format:

```
EchoTime() expression
```

3956 The following examples use EchoTime() to determine how long it takes to add the
3957 numbers 2 and 3 together and how long it takes to factor 1234567:

```
3958 In> EchoTime() 2 + 3
3959 Result: 5
3960 Side Effects:
3961 0.000080946 seconds taken.
3962
3962 In> EchoTime() Factor(1234567)
3963 Result: 127*9721
3964 Side Effects:
3965 0.395028773 seconds taken.
```

3966 In the following program, a ForEach loop is used to have the Factor() function

3967 factor all the numbers in a list. The EchoTime() function is used to determine
3968 how long it takes to do all the factoring:

```
3969 %mathpiper
3970
3971 EchoTime() ForEach(number, {100, 54, 65, 67, 344, 98, 454})
3972 [
3973     Echo(number, " - ", Factor(number));
3974 ];
3975
3976 %/mathpiper
3977
3978 %output,preserve="false"
3979 Result: True
3980
3981 Side Effects:
3982 100 - 2^2*5^2
3983 54 - 2*3^3
3984 65 - 5*13
3985 67 - 67
3986 344 - 2^3*43
3987 98 - 2*7^2
3988 454 - 2*227
3989 0.262678978 seconds taken.
3990 . %/output
```

3990 Finally, the following program show how to time a code block which prints the
3991 numbers from 1 to 100:

```
3992 %mathpiper
3993
3994 EchoTime()
3995 [
3996     index := 1;
3997
3998     While(index <= 100)
3999     [
4000         Write(index,,);
4001
4002         If(index % 10 = 0, NewLine());
4003
4004         index++;
4005     ];
4006
4007     NewLine();
4008 ];
4009 %/mathpiper
```



```
4010 %output,preserve="false"
4011 Result: True
4012
4013 Side Effects:
4014 1,2,3,4,5,6,7,8,9,10,
4015 11,12,13,14,15,16,17,18,19,20,
4016 21,22,23,24,25,26,27,28,29,30,
4017 31,32,33,34,35,36,37,38,39,40,
4018 41,42,43,44,45,46,47,48,49,50,
4019 51,52,53,54,55,56,57,58,59,60,
4020 61,62,63,64,65,66,67,68,69,70,
4021 71,72,73,74,75,76,77,78,79,80,
4022 81,82,83,84,85,86,87,88,89,90,
4023 91,92,93,94,95,96,97,98,99,100,
4024
4025 0.055418423 seconds taken.
4026 . %/output
```

4027 19.2 Exercises

4028 For the following exercises, create a new MathRider worksheet file called
4029 **book_1_section_19_exercises_<your first name>_<your last name>.mrw.**
4030 (**Note: there are no spaces in this file name**). For example, John Smith's
4031 worksheet would be called:

4032 **book_1_section_19_exercises_john_smith.mrw.**

4033 After this worksheet has been created, place your answer for each exercise that
4034 requires a fold into its own fold in this worksheet. Place a title attribute in the
4035 start tag of each fold which indicates the exercise the fold contains the solution
4036 to. The folds you create should look similar to this one:

```
4037 %mathpiper,title="Exercise 1"
```

```
4038 //Sample fold.
```

```
4039 %/mathpiper
```

4040 If an exercise uses the MathPiper console instead of a fold, copy the work you
4041 did in the console into the worksheet so it can be saved but do not put it in a fold.

4042 19.2.1 Exercise 1

4043 Create a program which uses a While loop to display the numbers from 1 to
4044 50. Use the ++ operator to increment the loop index variable.

4045 19.2.2 Exercise 2

4046 Create a program which uses a While loop to display the numbers from 1 to

4047 50 in reverse order. Use the -- operator to decrement the loop index
4048 variable.

4049 **19.2.3 Exercise 3**

4050 Create a program which uses a Break() function to cause a While loop which
4051 is configured to print the numbers from 1 to 100 to skip printing the
4052 number 72.

4053 **19.2.4 Exercise 4**

4054 Create a program which uses the version of the Repeat() function which
4055 takes an integer as an argument and the : string concatenation operator to
4056 print the following:

4057 Hello
4058 HelloHello
4059 HelloHelloHello
4060 HelloHelloHelloHello
4061 HelloHelloHelloHelloHello

4062 **19.2.5 Exercise 5**

4063 In the last example in the EchoTime() section, what operator is being used
4064 to format the output into lines of 10 numbers and how is this operator
4065 doing this?