

MathRider For Newbies

by Ted Kosan

Copyright © 2009 by Ted Kosan

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

Table of Contents

1	Preface.....	8
1.1	Dedication.....	8
1.2	Acknowledgments.....	8
1.3	Support Email List.....	8
2	Introduction.....	9
2.1	What Is A Mathematics Computing Environment?.....	9
2.2	What Is MathRider?.....	10
2.3	What Inspired The Creation Of Mathrider?.....	11
3	Downloading And Installing MathRider.....	13
3.1	Installing Sun's Java Implementation.....	13
3.1.1	Installing Java On A Windows PC.....	13
3.1.2	Installing Java On A Macintosh.....	13
3.1.3	Installing Java On A Linux PC.....	13
3.2	Downloading And Extracting.....	14
3.2.1	Extracting The Archive File For Windows Users.....	15
3.2.2	Extracting The Archive File For Unix Users.....	15
3.3	MathRider's Directory Structure & Execution Instructions.....	15
3.3.1	Executing MathRider On Windows Systems.....	16
3.3.2	Executing MathRider On Unix Systems.....	16
3.3.2.1	MacOS X.....	16
4	The Graphical User Interface.....	17
4.1	Buffers And Text Areas.....	17
4.2	The Gutter.....	17
4.3	Menus.....	17
4.3.1	File.....	18
4.3.2	Edit.....	18
4.3.3	Search.....	18
4.3.4	Markers, Folding, and View.....	19
4.3.5	Utilities.....	19
4.3.6	Macros.....	19
4.3.7	Plugins.....	19
4.3.8	Help.....	19
4.4	The Toolbar.....	19
5	MathPiper: A Computer Algebra System For Beginners.....	21
5.1	Numeric Vs. Symbolic Computations.....	21
5.2	Using The MathPiper Console As A Numeric (Scientific) Calculator.....	22
5.2.1	Functions.....	23

5.2.2 Accessing Previous Input And Results.....	24
5.2.3 Syntax Errors.....	24
5.3 Using The MathPiper Console As A Symbolic Calculator.....	25
5.3.1 Variables.....	25
5.3.1.1 Calculating With Unbound Variables.....	26
5.3.1.2 Variable And Function Names Are Case Sensitive.....	28
5.3.1.3 Using More Than One Variable.....	28
5.4 Exercises.....	29
5.4.1 Exercise 1.....	29
5.4.2 Exercise 2.....	29
5.4.3 Exercise 3.....	29
6 The MathPiper Documentation Plugin.....	31
6.1 Function List.....	31
6.2 Mini Web Browser Interface.....	31
6.3 Exercises.....	32
6.3.1 Exercise 1.....	32
6.3.2 Exercise 2.....	32
7 Using MathRider As A Programmer's Text Editor.....	33
7.1 Creating, Opening, Saving, And Closing Text Files.....	33
7.2 Editing Files.....	33
7.3 File Modes.....	33
7.4 Exercises.....	34
7.4.1 Exercise 1.....	34
8 MathRider Worksheet Files.....	35
8.1 Code Folds.....	35
8.1.1 The Description Attribute.....	36
8.2 Automatically Inserting Folds & Removing Unpreserved Folds.....	36
8.3 Exercises.....	37
8.3.1 Exercise 1.....	37
8.3.2 Exercise 2.....	37
8.3.3 Exercise 3.....	37
8.3.4 Exercise 4.....	37
9 MathPiper Programming Fundamentals.....	38
9.1 Values and Expressions.....	38
9.2 Operators.....	38
9.3 Operator Precedence.....	39
9.4 Changing The Order Of Operations In An Expression.....	40
9.5 Functions & Function Names.....	41
9.6 Functions That Produce Side Effects.....	42
9.6.1 The Echo(), Write(), and NewLine() Functions.....	42
9.6.1.1 Echo().....	42

9.6.1.2 Write().....	44
9.6.1.3 NewLine().....	45
9.7 Expressions Are Separated By Semicolons.....	46
9.7.1 Placing More Than One Expression On A Line In A Fold.....	46
9.7.2 Placing More Than One Expression On A Line In The Console Using A Code Block.....	47
9.8 Strings.....	48
9.8.1 The MathPiper Console and MathPiper Folds Can Access The Same Variables.....	48
9.8.2 Using Strings To Make Echo's Output Easier To Read.....	48
9.8.3 Accessing The Individual Letters In A String.....	49
9.9 Comments.....	50
9.10 Exercises.....	51
9.10.1 Exercise 1.....	51
9.10.2 Exercise 2.....	51
9.10.3 Exercise 3.....	51
9.10.4 Exercise 4.....	52
9.10.5 Exercise 5.....	52
9.10.6 Exercise 6.....	52
10 Rectangular Selection Mode And Text Area Splitting.....	54
10.1 Rectangular Selection Mode.....	54
10.2 Text area splitting.....	54
11 Working With Random Integers.....	56
11.1 Obtaining Nonnegative Random Integers With The RandomInteger() Function.....	56
11.2 Simulating The Rolling Of Dice.....	58
12 Making Decisions.....	59
12.1 Conditional Operators.....	59
12.2 Predicate Expressions.....	62
12.3 Exercises.....	62
12.3.1 Exercise 1.....	62
12.3.2 Exercise 2.....	63
12.4 Making Decisions With The If() Function & Predicate Expressions.....	63
12.4.1 If() Functions Which Include An "Else" Parameter.....	64
12.5 Exercises.....	65
12.5.1 Exercise 1.....	65
12.6 The And(), Or(), & Not() Boolean Functions & Infix Notation.....	65
12.6.1 And().....	65
12.6.2 Or().....	67
12.6.3 Not() & Prefix Notation.....	68
12.7 Exercises.....	69

12.7.1 Exercise 1.....	69
12.7.2 Exercise 2.....	70
13 The While() Looping Function & Bodied Notation.....	71
13.1 Printing The Integers From 1 to 10.....	71
13.2 Printing The Integers From 1 to 100.....	73
13.3 Printing The Odd Integers From 1 To 99.....	73
13.4 Printing The Integers From 1 To 100 In Reverse Order.....	74
13.5 Expressions Inside Of Code Blocks Are Indented.....	75
13.6 Long-Running Loops, Infinite Loops, & Interrupting Execution.....	75
13.7 A Program That Simulates Rolling Two Dice 50 Times.....	76
13.8 Exercises.....	78
13.8.1 Exercise 1.....	78
13.8.2 Exercise 2.....	78
13.8.3 Exercise 3.....	79
14 Predicate Functions.....	80
14.1 Finding Prime Numbers With A Loop.....	81
14.2 Finding The Length Of A String With The Length() Function.....	83
14.3 Converting Numbers To Strings With The String() Function.....	84
14.4 Finding Prime Numbers Which End With 7 (And Multi-line Function Calls)	84
14.5 Exercises.....	86
14.5.1 Exercise 1.....	86
14.5.2 Exercise 2.....	86
15 Lists: Values That Hold Sequences Of Expressions.....	87
15.1 Append() & Nondestructive List Operations.....	88
15.2 Using While Loops With Lists	89
15.2.1 Using A While Loop And Append() To Place Values In A List.....	91
15.3 Exercises.....	92
15.3.1 Exercise 1.....	92
15.3.2 Exercise 2.....	92
15.3.3 Exercise 3.....	92
15.4 The ForEach() Looping Function.....	92
15.5 Print All The Values In A List Using A ForEach() function.....	92
15.6 Calculate The Sum Of The Numbers In A List Using ForEach().....	93
15.7 The .. Range Operator.....	94
15.8 Using ForEach() With The Range Operator To Print The Prime Numbers Between 1 And 100.....	95
15.8.1 Using ForEach() And The Range Operator To Place The Prime Numbers Between 1 And 50 Into A List.....	96
15.8.2 Exercises.....	97
15.8.3 Exercise 1.....	97

15.8.4 Exercise 2.....	97
15.8.5 Exercise 3.....	97
15.8.6 Exercise 4.....	97
16 Functions & Operators Which Loop Internally.....	98
16.1 Functions & Operators Which Loop Internally To Process Lists.....	98
16.1.1 TableForm().....	98
16.1.2 Contains().....	98
16.1.3 Find().....	99
16.1.4 Count().....	99
16.1.5 Select().....	100
16.1.6 The Nth() Function & The [] Operator.....	100
16.1.7 The : Prepend Operator.....	101
16.1.8 Concat().....	101
16.1.9 Insert(), Delete(), & Replace().....	101
16.1.10 Take()	102
16.1.11 Drop().....	103
16.1.12 FillList().....	103
16.1.13 RemoveDuplicates().....	104
16.1.14 Reverse().....	104
16.1.15 Partition().....	104
16.1.16 Table()	105
16.2 Functions That Work With Integers.....	106
16.2.1 RandomIntegerVector().....	106
16.2.2 Max() & Min().....	106
16.2.3 Div() & Mod().....	107
16.2.4 Gcd().....	108
16.2.5 Lcm().....	108
16.2.6 Add().....	108
16.2.7 Factorize().....	109
16.3 Exercises.....	109
16.3.1 Exercise 1.....	109
16.3.2 Exercise 2.....	109
16.3.3 Exercise 3.....	110
16.3.4 Exercise 4.....	110
16.3.5 Exercise 5.....	110
17 Nested Loops.....	111
17.1 Generate All The Combinations That Can Be Entered Into A Two Digit Wheel Lock Using Two Nested Loops.....	111
17.2 Exercises.....	112
17.2.1 Exercise 1.....	112
18 User Defined Functions.....	114

18.1 Global Variables, Local Variables, & Local().....	116
18.2 Exercises.....	117
18.2.1 Exercise 1.....	117
18.2.2 Exercise 2.....	118
19 Models.....	119
19.1 Placing Models Into A Computer.....	120

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>. Please place **[Newbies book]**
15 in the title of your email when you post to this list if the topic of the post is
16 related to this book.

17 **2 Introduction**

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

27 **2.1 What Is A Mathematics Computing Environment?**

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

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

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

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

45 Some environments are highly specialized and some are general purpose. Some
46 allow mathematics to be entered and displayed in traditional form (which is what
47 is found in most math textbooks). Some are able to display traditional form
48 mathematics but need to have it input as text and some are only able to have
49 mathematics displayed and entered as text.

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

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

52 and here is the same formula in text form:

53
$$a = x^2 + 4 \cdot h \cdot x + 3/7$$

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

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

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

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

75 **2.2 What Is MathRider?**

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

81 MathRider uses MathPiper as its default computer algebra system, BeanShell as
82 its main scripting language, jEdit as its framework (hereafter referred to as the
83 MathRider framework), and Java as its overall implementation language. One
84 way to determine a person's MathRider expertise is by their knowledge of these
85 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.

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

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

93 **2.3 What Inspired The Creation Of Mathrider?**

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

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

97 and Steve Yegge's thoughts on learning mathematics:

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

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

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

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

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

110 **3 Downloading And Installing MathRider**

111 **3.1 *Installing Sun's Java Implementation***

112 MathRider is a Java-based application and therefore a current version of Sun's
113 Java (at least Java 5) must be installed on your computer before MathRider can
114 be run. (Note: If you cannot get Java to work on your system, some versions of
115 MathRider include Java in the download file and these files will have "with_java"
116 in their file names.)

117 **3.1.1 Installing Java On A Windows PC**

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

121 <http://java.com/>

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

124 **3.1.2 Installing Java On A Macintosh**

125 Macintosh computers have Java pre-installed but you may need to upgrade to a
126 current version of Java (at least Java 5) before running MathRider. If you need
127 to update your version of Java, visit the following website:

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

129 **3.1.3 Installing Java On A Linux PC**

130 Traditionally, installing Sun's Java on a Linux PC has not been an easy process
131 because Sun's version of Java was not open source and therefore the major Linux
132 distributions were unable to distribute it. In the fall of 2006, Sun made the
133 decision to release their Java implementation under the GPL in order to help
134 solve problems like this. Unfortunately, there were parts of Sun's Java that Sun
135 did not own and therefore these parts needed to be rewritten from scratch
136 before 100% of their Java implementation could be released under the GPL.

137 As of summer 2008, the rewriting work is not quite complete yet, although it is
138 close. If you are a Linux user who has never installed Sun's Java before, this
139 means that you may have a somewhat challenging installation process ahead of
140 you.

141 You should also be aware that a number of Linux distributions distribute a non-
142 Sun implementation of Java which is not 100% compatible with it. Running

sophisticated GUI-based Java programs on a non-Sun version of Java usually does not work. In order to check to see what version of Java you have installed (if any), execute the following command in a shell (MathRider needs at least Java 5):

```
java -version
```

Currently, the MathRider project has the following two options for people who need to install Sun's Java:

- 1) Locate the Java documentation for your Linux distribution and carefully follow the instructions provided for installing Sun's Java on your system.
- 2) Download a version of MathRider that includes its own copy of the Java runtime (when one is made available).

3.2 Downloading And Extracting

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

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

<http://mathrider.org>

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

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

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

The archive file can be easily copied to a CD or USB drive if you would like to

180 install MathRider on another computer or give it to a friend.

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

184 3.2.1 Extracting The Archive File For Windows Users

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

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

197 3.2.2 Extracting The Archive File For Unix Users

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

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

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

204 3.3 MathRider's Directory Structure & Execution Instructions

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

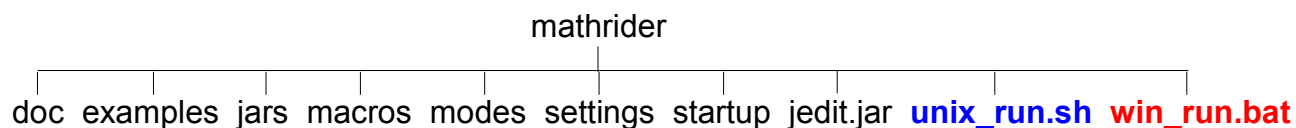


Illustration 1: MathRider's Directory Structure

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

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

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

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

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

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

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

215 **startup** - Contains startup scripts that are executed each time MathRider
216 launches.

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

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

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

220 **3.3.1 Executing MathRider On Windows Systems**

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

223 **3.3.2 Executing MathRider On Unix Systems**

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

226 `sh unix_run.sh`

227 **3.3.2.1 MacOS X**

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

231 `cd /Applications/mathrider`

232 Run mathrider by typing:

233 `sh unix_run.sh`

234 4 The Graphical User Interface

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

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

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

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

254 4.1 Buffers And Text Areas

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

261 4.2 The Gutter

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

265 4.3 Menus

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

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

272 4.3.1 File

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

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

278 4.3.2 Edit

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

284 4.3.3 Search

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

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

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

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

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

307 contents of the **Search for** text area and replace them with the contents of the
308 **Replace with** text area.

309 **4.3.4 Markers, Folding, and View**

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

311 **4.3.5 Utilities**

312 The utilities menu contains a significant number of actions, some that are useful
313 to beginners and others that are meant for experts. The two actions that are
314 most useful to beginners are the **Buffer Options** actions and the **Global**
315 **Options** actions. The **Buffer Options** actions allows the currently selected
316 buffer to be customized and the **Global Options** actions brings up a rich dialog
317 window that allows numerous aspects of the MathRider application to be
318 configured.

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

320 **4.3.6 Macros**

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

322 **4.3.7 Plugins**

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

330 **4.3.8 Help**

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

334 **4.4 The Toolbar**

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

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

342 **5 MathPiper: A Computer Algebra System For Beginners**

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

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

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

360 **5.1 Numeric Vs. Symbolic Computations**

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

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

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

378 **5.2 Using The MathPiper Console As A Numeric (Scientific) Calculator**

379 Open the Console plugin by selecting the **Console** tab in the lower left part of
380 the MathRider application. A text area will appear and in the upper left corner
381 of this text area will be a pull down menu which may be set to "MathPiper" or to
382 "System". If it is set to System, select this pull down menu and then select the
383 **MathPiper** menu item that is inside of it. Feel free to increase the size of the
384 console text area if you would like by dragging on the dotted lines which are at
385 the top side and right side of the console window.

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

388 MathPiper version ".75a".

389 In>

390 Click to the right of the prompt in order to place the cursor there then type **2+2**
391 followed by **<enter>**:

392 In> 2+2

393 Result> 4

394 In>

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

401 In addition to addition, MathPiper can also do subtraction, multiplication,
402 exponents, and division:

403 In> 5-2

404 Result> 3

405 In> 3*4

406 Result> 12

407 In> 2^3

408 Result> 8

409 In> 12/6

410 Result> 2

411 Notice that the multiplication symbol is an asterisk (*), the exponent symbol is a
412 caret (^), and the division symbol is a forward slash (/). These symbols (along
413 with addition (+) , subtraction (−), and ones we will talk about later) are called
414 **operators** because they tell MathPiper to perform an operation such as addition
415 or division.

416 MathPiper can also work with decimal numbers:

```
417 In> .5+1.2  
418 Result> 1.7
```

```
419 In> 3.7-2.6  
420 Result> 1.1
```

```
421 In> 2.2*3.9  
422 Result> 8.58
```

```
423 In> 2.2^3  
424 Result> 10.648
```

```
425 In> 9.5/3.2  
426 Result> 9.5/3.2
```

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

```
430 In> N(9.5/3.2)  
431 Result> 2.96875
```

432 As can be seen here, when a result is given in numeric form, it means that it is
433 given as a decimal number.

434 5.2.1 Functions

435 **N()** is an example of a **function**. A function can be thought of as a "black box"
436 which accepts input, processes the input, and returns a result. Each function
437 has a name and in this case, the name of the function is **N** which stands for
438 **Numeric**. To the right of a function's name there is always a set of parentheses
439 and information that is sent to the function is placed inside of them. The purpose
440 of the **N()** function is to make sure that the information that is sent to it is
441 processed numerically instead of symbolically.

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

```
444 In> IsEven(4)  
445 Result> True
```

```
446 In> IsEven(5)
447 Result> False
```

448 MathPiper has a large number of functions some of which are described in more
449 depth in the MathPiper Documentation section and the MathPiper Programming
450 Fundamentals section. **A complete list of MathPiper's functions is**
451 **contained in the MathPiperDocs plugin and more of these functions will**
452 **be discussed soon.**

453 5.2.2 Accessing Previous Input And Results

454 The MathPiper console keeps a history of all input lines that have been entered.
455 If the **up arrow** near the lower right of the keyboard is pressed, each previous
456 input line is displayed in turn to the right of the current input prompt.

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

```
460 In> 5*8
461 Result> 40

462 In> %
463 Result> 40

464 In> %*2
465 Result> 80
```

466 5.2.3 Syntax Errors

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

```
472 In> 12 \ 6
473 Error parsing expression, near token \
```

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

477 This section provided a short introduction to using MathPiper as a numeric
478 calculator and the next section contains a short introduction to using MathPiper

479 as a symbolic calculator.

480 **5.3 Using The MathPiper Console As A Symbolic Calculator**

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

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

```
486 In> 1/2 + 1/3  
487 Result> 5/6
```

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

```
492 In> %  
493 Result> 5/6
```

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

```
496 In> Numer(%)  
497 Result> 5
```

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

501 **5.3.1 Variables**

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

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

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

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

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

```
519 In> Numer(a)  
520 Result> 5
```

```
521 In> Denom(a)  
522 Result> 6
```

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

530 **5.3.1.1 Calculating With Unbound Variables**

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

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

```
534 In> a  
535 Result> 9
```

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

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

```
540 In> a
```

541 `Result> a`

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

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

551 `In> Factor(8)`

552 `Result> 2^3`

553 `In> Factor(14)`

554 `Result> 2*7`

555 `In> Factor(2343)`

556 `Result> 3*11*71`

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

558 `In> x`

559 `Result> x`

560 `In> IsBound(x)`

561 `Result> False`

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

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

564 `In> Expand(%)`

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

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

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

578 **5.3.1.2 Variable And Function Names Are Case Sensitive**

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

```
585 In> Box := 1
586 Result> 1
```

```
587 In> box := 2
588 Result> 2
```

```
589 In> Box
590 Result> 1
```

```
591 In> box
592 Result> 2
```

593 **5.3.1.3 Using More Than One Variable**

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

```
596 a := 2
597 Result> 2
```

```
598 b := 3
599 Result> 3
```

```
600 a + b
601 Result> 5
```

```
602 answer := a + b
603 Result> 5
```

```
604 answer
605 Result> 5
```

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

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

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

614 **5.4 Exercises**

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

617 **5.4.1 Exercise 1**

618 Perform the following calculation:

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

619 **5.4.2 Exercise 2**

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

622 In> ans := 1/5 + 7/4 + 15/16

623 b) Use the Numer() function to calculate the numerator of ans.

624 c) Use the Denom() function to calculate the denominator of ans.

625 d) Use the N() function to calculate the numeric value of ans.

626 e) Use the Clear() function to unbind the variable ans and verify that ans
627 is unbound by executing the following code and by using the IsBound()
628 function:

629 In> ans

630 **5.4.3 Exercise 3**

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

633 a)

634 In> x

635 b)

636 In> y

637 c)

638 In> z

639 d)

640 In> x + y

641 d)

642 In> x + z

643 e)

644 In> x + y + z

645 **6 The MathPiper Documentation Plugin**

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

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

654 **6.1 Function List**

655 MathPiper's functions are divided into two main categories called **user** functions
656 and **programmer functions**. In general, the **user functions** are used for
657 solving problems in the MathPiper console or with short programs and the
658 **programmer functions** are used for longer programs. However, users will
659 often use some of the programmer functions and programmers will use the user
660 functions as needed.

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

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

678 **6.2 Mini Web Browser Interface**

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

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

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

689 **6.3 Exercises**

690 **6.3.1 Exercise 1**

691 Locate the `N()`, `IsEven()`, `IsOdd()`, `Clear()`, `IsBound()`, `Numer()`, `Denom()`,
692 and `Factor()` functions in the **All Functions** section of the MathPiperDocs
693 plugin and read the information that is available on them.

694 **6.3.2 Exercise 2**

695 Locate the `N()`, `IsEven()`, `IsOdd()`, `Clear()`, `IsBound()`, `Numer()`, `Denom()`,
696 and `Factor()` functions in the **User Functions** section of the MathPiperDocs
697 plugin and list which section each function is contained in. Don't include
698 the **Alphabetical** or **Built In** subsections in your search.

699 **7 Using MathRider As A Programmer's Text Editor**

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

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

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

713 **7.1 Creating, Opening, Saving, And Closing Text Files**

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

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

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

726 **7.2 Editing Files**

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

732 **7.3 File Modes**

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

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

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

742 **7.4 Exercises**

743 **7.4.1 Exercise 1**

744 Create a text file called "**my_text_file.txt**" and place a few sentences in
745 it. Save the text file somewhere on your hard drive then close it. Now,
746 open the text file again using **File->Open** and verify that what you typed is
747 still in the file.

748 8 MathRider Worksheet Files

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

756 8.1 Code Folds

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

764 For example, here is a MathPiper fold which will print the result of **2 + 3** to the
765 MathPiper console (**Note: the line numbers are not part of the program and**
766 **the semicolon ';' which is at the end of the line of code is required**):

```
767 1:%mathpiper  
768 2:  
769 3:2 + 3;  
770 4:  
771 5:%/mathpiper
```

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

```
776 1:%mathpiper  
777 2:  
778 3:2 + 3;  
779 4:  
780 5:%/mathpiper  
781 6:  
782 7:    %output,preserve="false"  
783 8:    Result: 5  
784 9:    %/output
```

785 The default type of an output fold is **%output** and this one starts at **line 7** and

ends on **line 9**. Folds that can be executed have their first and last lines highlighted and folds that cannot be executed do not have their first and last lines highlighted. By default, folds of type %output have their **preserve property** set to **false**. This tells MathRider to overwrite the %output fold with a new version during the next execution of its parent.

8.1.1 The Description Attribute

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

```
1:%mathpiper,description="Add two numbers together."  
2:  
3:2 + 3;  
4:  
5:%/mathpiper
```

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

8.2 Automatically Inserting Folds & Removing Unpreserved Folds

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

```
%mathpiper  
  
%/mathpiper
```

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

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

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

819 **8.3 Exercises**

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

822 [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)
823 [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)

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

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

831 **8.3.1 Exercise 1**

832 Execute folds 1-8 in the top section of the worksheet by placing the cursor
833 inside of the fold and then pressing <shift><enter> on the keyboard.

834 **8.3.2 Exercise 2**

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

837 **8.3.3 Exercise 3**

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

839 **8.3.4 Exercise 4**

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

843 a) $2 \cdot 7 + 3$

844 b) $18/3$

845 c) $234238342 + 2038408203$

846 d) $324802984 \cdot 2308098234$

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

848 9 MathPiper Programming Fundamentals

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

853 9.1 Values and Expressions

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

856 3

857 represents the number three, the value:

858 0.5

859 represents the number one half, and the value:

860 "Mathematics is powerful!"

861 represents an English sentence.

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

865 3

866 2 + 3

867 5 + 6*21/18 - 2^3

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

871 In> 2 + 3

872 Result> 5

873 9.2 Operators

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

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

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

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

884 In> 5 - 2
885 Result> 3

886 In> 3*4
887 Result> 12

888 In> 30/3
889 Result> 10

890 In> 8%5
891 Result> 3

892 In> 2^3
893 Result> 8

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

895 In> -3
896 Result> -3

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

899 In> - -3
900 Result> 3

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

906 **9.3 Operator Precedence**

907 When expressions contain more than one operator, MathPiper uses a set of rules
908 called **operator precedence** to determine the order in which the operators are
909 applied to the values in the expression. Operator precedence is also referred to
910 as the **order of operations**. Operators with higher precedence are evaluated
911 before operators with lower precedence. The following table shows a subset of
912 MathPiper's operator precedence rules with higher precedence operators being
913 placed higher in the table:

914 [^] Exponents are evaluated right to left.

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

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

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

920 5 + 6*21/18 - 2^3

921 And here it is in traditional form:

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

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

924 5 + 6*21/18 - 2^3

925 5 + 6*21/18 - 8

926 5 + 126/18 - 8

927 5 + 7 - 8

928 12 - 8

929 4

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

934 **9.4 Changing The Order Of Operations In An Expression**

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

940 In> 2 + 4*5

941 Result> 22

942 If parentheses are placed around 4 + 5, however, the addition operator is forced
943 to be evaluated before the multiplication operator and the result is 30:


```
944 In> (2 + 4)*5
945 Result> 30
```

946 Parentheses can also be nested and nested parentheses are evaluated from the
947 most deeply nested parentheses outward:

```
948 In> ((2 + 4)*3)*5
949 Result> 90
```

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

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

- 954 () Parentheses are evaluated from the inside out.
- 955 ^ Then exponents are evaluated right to left.
- 956 *,%,/ Then multiplication, remainder, and division operations are evaluated
957 left to right.
- 958 +, - Finally, addition and subtraction are evaluated left to right.

959 **9.5 Functions & Function Names**

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

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

970 The functions that come with MathPiper have names which consist of either a
971 single word (such as **Add()**) or multiple words that have been put together to
972 form a compound word (such as **IsBound()**). All letters in the names of
973 functions which come with MathPiper are lower case except the beginning letter
974 in each word, which are upper case.

975 **9.6 Functions That Produce Side Effects**

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

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

986 **9.6.1 The Echo(), Write(), and NewLine() Functions**

987 The **Echo()** and **Write()** functions both send information to the user and this is
988 often referred to as "printing" in this document. It may also be called "echoing"
989 and "writing".

990 **9.6.1.1 Echo()**

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

```
994 In> Echo(1)
995 Result> True
996 Side Effects>
997 1
```

998 In this example, the number 1 was passed to the Echo() function, the number
999 was evaluated (all numbers evaluate to themselves), and the result of the
1000 evaluation was then printed as a side effect. Notice that Echo() **also returned a**
1001 **result**. In MathPiper, all functions return a result, but functions whose main
1002 purpose is to produce a side effect usually just return a result of **True** if the side
1003 effect succeeded or **False** if it failed. In this case, Echo() returned a result of
1004 **True** because it was able to successfully print a 1 as its side effect.

1005 The next example shows multiple expressions being sent to Echo() (notice that
1006 the expressions are separated by commas):

```
1007 In> Echo(1,1+2,2*3)
1008 Result> True
1009 Side Effects>
1010 1 3 6
```

1011 The expressions were each evaluated and their results were returned (separated
1012 by spaces) as side effect output. If it is desired that commas be printed between
1013 the numbers in the output, simply place three commas between the expressions
1014 that are passed to Echo():

```
1015 In> Echo(1,,,1+2,,,2*3)
1016 Result> True
1017 Side Effects>
1018 1 , 3 , 6
```

1019 Each time an Echo() function is executed, it always forces the display to drop
1020 down to the next line after it is finished. This can be seen in the following
1021 program which is similar to the previous one except it uses a separate Echo()
1022 function to display each expression:

```
1023 1:%mathpiper
1024 2:
1025 3:Echo(1);
1026 4:
1027 5:Echo(1+2);
1028 6:
1029 7:Echo(2*3);
1030 8:
1031 9:%/mathpiper
1032 10:
1033 11:    %output,preserve="false"
1034 12:    Result: True
1035 13:
1036 14:    Side Effects:
1037 15:    1
1038 16:    3
1039 17:    6
1040 18:    %/output
```

1041 Notice how the 1, the 3, and the 6 are each on their own line.

1042 Now that we have seen how Echo() works, lets use it to do something useful. If
1043 more than one expression is evaluated in a %mathpiper fold, only the result from
1044 the last expression that was evaluated (which is usually the bottommost
1045 expression) is displayed:

```
1046 1:%mathpiper
1047 2:
1048 3:a := 1;
1049 4:b := 2;
1050 5:c := 3;
1051 6:
1052 7:%/mathpiper
1053 8:
```

```
1054 9:      %output,preserve="false"
1055 10:      Result: 3
1056 11:      %/output
```

1057 In MathPiper, programs are executed one line at a time, starting at the topmost
1058 line of code and working downwards from there. In this example, the line `a := 1;`
1059 is executed first, then the line `b := 2;` is executed, and so on. Notice, however,
1060 that even though we wanted to see what was in all three variables, only the
1061 content of the last variable was displayed.

1062 The following example shows how `Echo()` can be used to display the contents of
1063 all three variables:

```
1064 1:%mathpiper
1065 2:
1066 3:a := 1;
1067 4:Echo(a);
1068 5:
1069 6:b := 2;
1070 7:Echo(b);
1071 8:
1072 9:c := 3;
1073 10:Echo(c);
1074 11:
1075 12:%/mathpiper
1076 13:
1077 14:      %output,preserve="false"
1078 15:      Result: True
1079 16:
1080 17:      Side Effects:
1081 18:      1
1082 19:      2
1083 20:      3
1084 21:      %/output
```

1085 9.6.1.2 Write()

1086 The **Write()** function is similar to the `Echo()` function except it does not
1087 automatically drop the display down to the next line after it finishes executing:

```
1088 1:%mathpiper
1089 2:
1090 3:Write(1);
1091 4:
1092 5:Write(1+2);
1093 6:
1094 7:Echo(2*3);
1095 8:
1096 9:%/mathpiper
```

```
1097 10:
1098 11:     %output,preserve="false"
1099 12:     Result: True
1100 13:
1101 14:     Side Effects:
1102 15:     1 3 6
1103 16:     %/output
```

1104 Write() and Echo() have other differences besides the one discussed here and
1105 more information about them can be found in the documentation for these
1106 functions.

1107 **9.6.1.3 NewLine()**

1108 The **NewLine()** function simply prints a blank line in the side effects output. It
1109 is useful for placing vertical space between printed lines:

```
1110 1: %mathpiper
1111 2:
1112 3: a := 1;
1113 4: Echo(a);
1114 5: NewLine();
1115 6:
1116 7: b := 2;
1117 8: Echo(b);
1118 9: NewLine();
1119 10:
1120 11: c := 3;
1121 12: Echo(c);
1122 13:
1123 14: %/mathpiper
1124 15:
1125 16:     %output,preserve="false"
1126 17:     Result: True
1127 18:
1128 19:     Side Effects:
1129 20:     1
1130 21:
1131 22:     2
1132 23:
1133 24:     3
1134 25:     %/output
```

1135 **9.7 Expressions Are Separated By Semicolons**

1136 As discussed earlier, all of the expressions that are inside of a **%mathpiper** fold
1137 must have a semicolon (;) after them. However, the expressions executed in the
1138 **MathPiper console** did not have a semicolon after them. MathPiper actually

1139 requires that all expressions end with a semicolon, but one does not need to add
1140 a semicolon to an expression which is typed into the MathPiper console **because**
1141 **the console adds it automatically** when the expression is executed.

1142 9.7.1 Placing More Than One Expression On A Line In A Fold

1143 All the previous code examples have had each of their expressions on a separate
1144 line, but multiple expressions can also be placed on a single line because the
1145 semicolons tell MathPiper where one expression ends and the next one begins:

```
1146 1:%mathpiper
1147 2:
1148 3:a := 1; Echo(a); b := 2; Echo(b); c := 3; Echo(c);
1149 4:
1150 5:%/mathpiper
1151 6:
1152 7:    %output,preserve="false"
1153 8:    Result: True
1154 9:
1155 10:    Side Effects:
1156 11:    1
1157 12:    2
1158 13:    3
1159 14:    %/output
```

1160 The spaces that are in the code on line 2 of this example are used to make the
1161 code more readable. Any spaces that are present within any expressions or
1162 between them are ignored by MathPiper and if we remove the spaces from the
1163 previous code, the output remains the same:

```
1164 1:%mathpiper
1165 2:
1166 3:a:=1;Echo(a);b:=2;Echo(b);c:= 3;Echo(c);
1167 4:
1168 5:%/mathpiper
1169 6:
1170 7:    %output,preserve="false"
1171 8:    Result: True
1172 9:
1173 10:    Side Effects:
1174 11:    1
1175 12:    2
1176 13:    3
1177 14:    %/output
```

1178 **9.7.2 Placing More Than One Expression On A Line In The Console Using** 1179 **A Code Block**

1180 The MathPiper console is only able to execute one expression at a time so if the
1181 previous code that executes three variable assignments and three Echo()
1182 functions on a single line is evaluated in the console, only the expression **a := 1**
1183 is executed:

```
1184 In> a := 1; Echo(a); b := 2; Echo(b); c := 3; Echo(c);  
1185 Result> 1
```

1186 Fortunately, this limitation can be overcome by placing the code into a **code**
1187 **block**. A **code block** (which is also called a **compound expression**) consists of
1188 one or more expressions which are separated by semicolons and placed within an
1189 open bracket (**[**) and close bracket (**]**) pair. If a code block is evaluated in the
1190 MathPiper console, each expression in the block will be executed from left to
1191 right. The following example shows the previous code being executed within of a
1192 code block inside the MathPiper console:

```
1193 In> [a := 1; Echo(a); b := 2; Echo(b); c := 3; Echo(c);]  
1194 Result> True  
1195 Side Effects>  
1196 1  
1197 2  
1198 3
```

1199 Notice that this time all of the expressions were executed and 1-3 was printed as
1200 a side effect. Code blocks always return the result of the last expression
1201 executed as the result of the whole block. In this case, True was returned as the
1202 result because the last Echo(c) function returned True. If we place another
1203 expression after the Echo(c) function, however, the block will execute this new
1204 expression last and its result will be the one returned by the block:

```
1205 In> [a := 1; Echo(a); b := 2; Echo(b); c := 3; Echo(c); 2 + 2]  
1206 Result> 4  
1207 Side Effects>  
1208 1  
1209 2  
1210 3
```

1211 Code blocks are very powerful and we will be discussing them further in later
1212 sections.

1213 **9.8 Strings**

1214 A **string** is a **value** that is used to hold text-based information. The typical

1215 expression that is used to create a string consists of **text which is enclosed**
1216 **within double quotes**. Strings can be assigned to variables just like numbers
1217 can and strings can also be displayed using the Echo() function. The following
1218 program assigns a string value to the variable 'a' and then echos it to the user:

```
1219 1:%mathpiper
1220 2:
1221 3:a := "Hello, I am a string.";
1222 4:Echo(a);
1223 5:
1224 6:%/mathpiper
1225 7:
1226 8:    %output,preserve="false"
1227 9:    Result: True
1228 10:
1229 11:    Side Effects:
1230 12:    Hello, I am a string.
1231 13:    %/output
```

1232 9.8.1 The MathPiper Console and MathPiper Folds Can Access The Same 1233 Variables

1234 A useful aspect of using MathPiper inside of MathRider is that variables that are
1235 assigned inside of a **%mathpiper fold** are accessible inside of the **MathPiper**
1236 **console** and variables that are assigned inside of the **MathPiper console** are
1237 available inside of **%mathpiper folds**. For example, after the above fold is
1238 executed, the string that has been bound to variable 'a' can be displayed in the
1239 MathPiper console:

```
1240 In> a
1241 Result> "Hello, I am a string."
```

1242 9.8.2 Using Strings To Make Echo's Output Easier To Read

1243 When the Echo() function is used to print the values of multiple variables, it is
1244 often helpful to print some information next to each variable so that it is easier to
1245 determine which value came from which Echo() function in the code. The
1246 following program prints the name of the variable that each value came from
1247 next to it in the side effects output:

```
1248 1:%mathpiper
1249 2:
1250 3:a := 1;
1251 4:Echo("Variable a: ", a);
1252 5:
1253 6:b := 2;
```



```
1254 7:Echo("Variable b: ", b);
1255 8:
1256 9:c := 3;
1257 10:Echo("Variable c: ", c);
1258 11:
1259 12:~/mathpiper
1260 13:
1261 14:    %output,preserve="false"
1262 15:    Result: True
1263 16:
1264 17:    Side Effects:
1265 18:    Variable a: 1
1266 19:    Variable b: 2
1267 20:    Variable c: 3
1268 21:    %/output
```

1269 9.8.3 Accessing The Individual Letters In A String

1270 Individual letters in a string (which are also called **characters**) can be accessed
1271 by placing the character's position number (also called an **index**) inside of
1272 brackets **[]** after the variable it is bound to. A character's position is determined
1273 by its distance from the left side of the string starting at 1. For example, in the
1274 string "Hello", 'H' is at position 1, 'e' is at position 2, etc. The following code
1275 shows individual characters in the above string being accessed:

```
1276 In> a := "Hello, I am a string."
1277 Result> "Hello, I am a string."

1278 In> a[1]
1279 Result> "H"

1280 In> a[2]
1281 Result> "e"

1282 In> a[3]
1283 Result> "l"

1284 In> a[4]
1285 Result> "l"

1286 In> a[5]
1287 Result> "o"
```

1288 9.9 Comments

1289 Source code can often be difficult to understand and therefore all programming
1290 languages provide the ability for **comments** to be included in the code.
1291 Comments are used to explain what the code near them is doing and they are

1292 usually meant to be read by humans instead of being processed by a computer.
1293 Therefore, comments are ignored by the computer when a program is executed.

1294 There are two ways that MathPiper allows comments to be added to source code.
1295 The first way is by placing two forward slashes `//` to the left of any text that is
1296 meant to serve as a comment. The text from the slashes to the end of the line
1297 the slashes are on will be treated as a comment. Here is a program that contains
1298 comments which use slashes:

```
1299 1:%mathpiper
1300 2://This is a comment.
1301 3:
1302 4:x := 2; //Set the variable x equal to 2.
1303 5:
1304 6:
1305 7:%/mathpiper
1306 8:
1307 9:     %output,preserve="false"
1308 10:     Result: 2
1309 11:     %/output
```

1310 When this program is executed, any text that starts with slashes is ignored.

1311 The second way to add comments to a MathPiper program is by enclosing the
1312 comments inside of slash-asterisk/asterisk-slash symbols `/* */`. This option is
1313 useful when a comment is too large to fit on one line. Any text between these
1314 symbols is ignored by the computer. This program shows a longer comment
1315 which has been placed between these symbols:

```
1316 1:%mathpiper
1317 2:
1318 3:/*
1319 4: This is a longer comment and it uses
1320 5: more than one line. The following
1321 6: code assigns the number 3 to variable
1322 7: x and then returns it as a result.
1323 8:*/
1324 9:
1325 10:x := 3;
1326 11:
1327 12:%/mathpiper
1328 13:
1329 14:     %output,preserve="false"
1330 15:     Result: 3
1331 16:     %/output
```

1332 9.10 Exercises

1333 For the following exercises, create a new MathRider worksheet file called
1334 **section_9_exercises_<your first name>_<your last name>.mrw**. (**Note:**
1335 **there are no spaces in this file name**). For example, John Smith's worksheet
1336 would be called:

1337 **section_9_exercises_john_smith.mrw**.

1338 After this worksheet has been created, place your answer for each exercise into
1339 its own fold in this worksheet. Place a description attribute in the start tag of
1340 each fold which indicates the exercise the fold contains the solution to. The folds
1341 you create should look similar to this one:

```
1342 1:%mathpiper,description="Exercise 1"  
1343 2:  
1344 3://Sample fold.  
1345 4:  
1346 5:%/mathpiper
```

1347 9.10.1 Exercise 1

1348 Change the precedence of the following expression using parentheses so that
1349 it prints 20 instead of 14:

1350 $2 + 3 * 4$

1351 9.10.2 Exercise 2

1352 Place the following calculations into a fold and then use one Echo()
1353 function per variable to print the results of the calculations. Put
1354 strings in the Echo() functions which indicate which variable each
1355 calculated value is bound to:

```
1356 a := 1+2+3+4+5;  
1357 b := 1-2-3-4-5;  
1358 c := 1*2*3*4*5;  
1359 d := 1/2/3/4/5;
```

1360 9.10.3 Exercise 3

1361 Place the following calculations into a fold and then use one Echo()
1362 function to print the results of all the calculations on a single line
1363 (Remember, the Echo() function can print multiple values if they are
1364 separated by commas.):

```
1365 Clear(x);  
1366 a := 2*2*2*2*2;  
1367 b := 2^5;
```

```
1368 c := x^2 * x^3;
1369 d := 2^2 * 2^3;
```

1370 9.10.4 Exercise 4

1371 The following code assigns a string which contains all of the upper case
1372 letters of the alphabet to the variable **upper**. Each of the three Echo()
1373 functions prints an index number and the letter that is at that position in
1374 the string. Place this code into a fold and then continue the Echo()
1375 functions so that all 26 letters and their index numbers are printed

```
1376 upper := "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

1377 Echo(1,upper[1]);
1378 Echo(2,upper[2]);
1379 Echo(3,upper[3]);
```

1380 9.10.5 Exercise 5

1381 Use Echo() functions to print an index number and the character at this
1382 position for the following string (this is similar to what was done in
1383 Exercise 4.):

```
1384 extra := "!.@#$$%^&*() _+<>,?/{ }[]|\-= ";

1385 Echo(1,extra[1]);
1386 Echo(2,extra[2]);
1387 Echo(3,extra[3]);
```

1388 9.10.6 Exercise 6

1389 The following program uses strings and index numbers to print a person's
1390 name. Create a program which uses the three strings from this program to
1391 print the names of three of your favorite movie actors.

```
1392 %mathpiper
1393 /*
1394    This program uses strings and index numbers to print
1395    a person's name.
1396 */

1397 upper := "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
1398 lower := "abcdefghijklmnopqrstuvwxyz";
1399 extra := "!.@#$$%^&*() _+<>,?/{ }[]|\-= ";

1400 //Print "Mary Smith.".
1401 Echo(upper[13],lower[1],lower[18],lower[25],extra[12],upper[19],lower[13],l
1402 ower[9],lower[20],lower[8],extra[1]);

1403 %/mathpiper
```

```
1404      %output,preserve="false"
1405      Result: True
1406
1407      Side Effects:
1408      Mary Smith.
1409 .    %/output
```

10 Rectangular Selection Mode And Text Area Splitting

10.1 Rectangular Selection Mode

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

- 1) Type 3 or 4 lines of text into a text area.
- 2) Hold down the **<Alt>** key then slowly press the **backslash key** (\) a few times. The bottom of the MathRider window contains a text field which MathRider uses to communicate information to the user. As **<Alt>** is repeatedly pressed, messages are displayed which read **Rectangular selection is on** and **Rectangular selection is off**.
- 3) Turn rectangular selection on and then select some text in order to see how this is different than normal selection mode. **When you are done experimenting, set rectangular selection mode to off.**

One thing that rectangular selection mode is very handy for is removing the line numbers from folds that are copied from this document into a MathRider worksheet. If you have been manually removing these line numbers with the arrow, delete, and backspace keys, I think you will find that using rectangular selection mode to remove them is much easier.

10.2 Text area splitting

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

MathRider has this ability and it is called **splitting**. If you look just to the right of the toolbar there is an icon which looks like a blank window, an icon to the right of it which looks like a window which was split horizontally, and an icon to the right of the horizontal one which is split vertically. If you let your mouse hover over these icons, a short description will be displayed for each of them. **(For now, ignore the icon which has a yellow sunburst on it. It is the New View icon and it is an advanced feature.)**

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

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 Nonnegative Random Integers With The RandomInteger() Function

One way that a MathPiper program can generate random integers is with the RandomInteger() function **(Note: the RandomInteger() function is not currently listed in the MathPiperDocs plugin.)** The RandomInteger() function takes an integer as a parameter and it returns a random integer between 0 and the passed in integer. The following example shows random integers between 0 and 4 **inclusive** being obtained from RandomInteger(). **Inclusive** here means that both 0 and 4 are included in the range of random integers that may be returned. If the word **exclusive** was used instead, this would mean that neither 0 nor 4 would be in the range.

```
In> RandomInteger(5)
Result> 3
In> RandomInteger(5)
Result> 4
In> RandomInteger(5)
Result> 3
In> RandomInteger(5)
Result> 1
In> RandomInteger(5)
Result> 2
In> RandomInteger(5)
Result> 4
In> RandomInteger(5)
Result> 1
In> RandomInteger(5)
Result> 1
In> RandomInteger(5)
Result> 0
In> RandomInteger(5)
Result> 1
```

If generating integers between 1 and 5 inclusive is desired instead of integers between 0 and 4, the number 1 can simply be added to the value which is returned by RandomInteger():

```
In> RandomInteger(5) + 1
```

```
1487 Result> 4
1488 In> RandomInteger(5) + 1
1489 Result> 5
1490 In> RandomInteger(5) + 1
1491 Result> 4
1492 In> RandomInteger(5) + 1
1493 Result> 2
1494 In> RandomInteger(5) + 1
1495 Result> 3
1496 In> RandomInteger(5) + 1
1497 Result> 5
1498 In> RandomInteger(5) + 1
1499 Result> 2
1500 In> RandomInteger(5) + 1
1501 Result> 2
1502 In> RandomInteger(5) + 1
1503 Result> 1
1504 In> RandomInteger(5) + 1
1505 Result> 2
```

1506 Random integers between 1 and 100 can be generated by passing 100 to
1507 RandomInteger() and adding 1 to the result.:

```
1508 In> RandomInteger(100) + 1
1509 Result> 15
1510 In> RandomInteger(100) + 1
1511 Result> 14
1512 In> RandomInteger(100) + 1
1513 Result> 82
1514 In> RandomInteger(100) + 1
1515 Result> 93
1516 In> RandomInteger(100) + 1
1517 Result> 32
```

1518 A range of random integers that does not start with 0 or 1 can also be generated.
1519 For example, random integers between 50 and 100 can be obtained by having
1520 RandomInteger() generate a random integer between 0 and 49 and then adding
1521 1 and 50 to the result:

```
1522 In> RandomInteger(50) + 1 + 50
1523 Result> 87
1524 In> RandomInteger(50) + 1 + 50
1525 Result> 100
1526 In> RandomInteger(50) + 1 + 50
1527 Result> 76
1528 In> RandomInteger(50) + 1 + 50
1529 Result> 60
1530 In> RandomInteger(50) + 1 + 50
1531 Result> 73
```


1532 **11.2 Simulating The Rolling Of Dice**

1533 The following example shows the simulated rolling of a single six sided die using
1534 the RandomInteger() function:

```
1535 In> RandomInteger(6) + 1
1536 Result> 5
1537 In> RandomInteger(6) + 1
1538 Result> 6
1539 In> RandomInteger(6) + 1
1540 Result> 3
1541 In> RandomInteger(6) + 1
1542 Result> 2
1543 In> RandomInteger(6) + 1
1544 Result> 5
```

1545 Code that simulates the rolling of two 6 sided dice can be evaluated in the
1546 MathPiper console by placing it within a **code block**. The following code
1547 outputs the sum of the two simulated dice:

```
1548 In> [a := RandomInteger(6) + 1; b := RandomInteger(6) + 1; a + b;]
1549 Result> 6
1550 In> [a := RandomInteger(6) + 1; b := RandomInteger(6) + 1; a + b;]
1551 Result> 12
1552 In> [a := RandomInteger(6) + 1; b := RandomInteger(6) + 1; a + b;]
1553 Result> 6
1554 In> [a := RandomInteger(6) + 1; b := RandomInteger(6) + 1; a + b;]
1555 Result> 4
1556 In> [a := RandomInteger(6) + 1; b := RandomInteger(6) + 1; a + b;]
1557 Result> 3
1558 In> [a := RandomInteger(6) + 1; b := RandomInteger(6) + 1; a + b;]
1559 Result> 8
```

1560 Now that we have the ability to simulate the rolling of two 6 sided dice, it would
1561 be interesting to determine if some sums of these dice occur more frequently
1562 than other sums. What we would like to do is to roll these simulated dice
1563 hundreds (or even thousands) of times and then analyze the sums that were
1564 produced. We don't have the programming capability to easily do this yet, but
1565 after we finish the section on while loops, we will.

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**. Table 2 shows the conditional operators that MathPiper uses:

Operator	Description
$x = y$	Returns True if the two values are equal and False if they are not equal. Notice that $=$ performs a comparison and not an assignment like $:=$ does.
$x \neq y$	Returns True if the values are not equal and False if they are equal.
$x < y$	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.
$x \leq y$	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.
$x > y$	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.
$x \geq y$	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

In> 4 > 5
Result> False
```

```
1585 In> 8 >= 8
1586 Result> True
```

```
1587 In> 5 <= 10
1588 Result> True
```

1589 The following examples show each of the conditional operators in Table 2 being
 1590 used to compare values that have been assigned to variables **x** and **y**:

```
1591 1:%mathpiper
1592 2:
1593 2:// Example 1.
1594 3:x := 2;
1595 4:y := 3;
1596 5:
1597 6:Echo(x, "=", y, ":", x = y);
1598 7:Echo(x, "!= ", y, ":", x != y);
1599 8:Echo(x, "< ", y, ":", x < y);
1600 9:Echo(x, "<=", y, ":", x <= y);
1601 10:Echo(x, "> ", y, ":", x > y);
1602 11:Echo(x, ">=", y, ":", x >= y);
1603 12:
1604 13:%/mathpiper
1605 14:
1606 15:    %output,preserve="false"
1607 16:    Result: True
1608 17:
1609 18:    Side Effects:
1610 19:    2 = 3 :False
1611 20:    2 != 3 :True
1612 21:    2 < 3 :True
1613 22:    2 <= 3 :True
1614 23:    2 > 3 :False
1615 24:    2 >= 3 :False
1616 25:    %/output
```

```
1617 1:%mathpiper
1618 2:
1619 3:    // Example 2.
1620 4:    x := 2;
1621 5:    y := 2;
1622 6:
1623 7:    Echo(x, "=", y, ":", x = y);
1624 8:    Echo(x, "!= ", y, ":", x != y);
1625 9:    Echo(x, "< ", y, ":", x < y);
1626 10:    Echo(x, "<=", y, ":", x <= y);
1627 11:    Echo(x, "> ", y, ":", x > y);
1628 12:    Echo(x, ">=", y, ":", x >= y);
1629 13:
1630 14:%/mathpiper
```

```

1631 15:
1632 16:     %output,preserve="false"
1633 17:     Result: True
1634 18:
1635 19:     Side Effects:
1636 20:     2 = 2 :True
1637 21:     2 != 2 :False
1638 22:     2 < 2 :False
1639 23:     2 <= 2 :True
1640 24:     2 > 2 :False
1641 25:     2 >= 2 :True
1642 25:     %/output

```

```

1643 1: %mathpiper
1644 2:
1645 3: // Example 3.
1646 4: x := 3;
1647 5: y := 2;
1648 6:
1649 7: Echo(x, "=", y, ":", x = y);
1650 8: Echo(x, "!= ", y, ":", x != y);
1651 9: Echo(x, "< ", y, ":", x < y);
1652 10: Echo(x, "<= ", y, ":", x <= y);
1653 11: Echo(x, "> ", y, ":", x > y);
1654 12: Echo(x, ">= ", y, ":", x >= y);
1655 13:
1656 14: %/mathpiper
1657 15:
1658 16:     %output,preserve="false"
1659 17:     Result: True
1660 18:
1661 19:     Side Effects:
1662 20:     3 = 2 :False
1663 21:     3 != 2 :True
1664 22:     3 < 2 :False
1665 23:     3 <= 2 :False
1666 24:     3 > 2 :True
1667 25:     3 >= 2 :True
1668 26:     %/output

```

1669 Conditional operators are placed at a lower level of precedence than the other
 1670 operators we have covered to this point:

- 1671 () Parentheses are evaluated from the inside out.
- 1672 ^ Then exponents are evaluated right to left.
- 1673 *,%,/ Then multiplication, remainder, and division operations are evaluated
 1674 left to right.

1675 +, − Then addition and subtraction are evaluated left to right.

1676 =,!=,<,<=,>,>= Finally, conditional operators are evaluated.

1677 **12.2 Predicate Expressions**

1678 Expressions which return either **True** or **False** are called "**predicate**"
1679 expressions. By themselves, predicate expressions are not very useful and they
1680 only become so when they are used with special decision making functions, like
1681 the If() function (which is discussed in the next section).

1682 **12.3 Exercises**

1683 **12.3.1 Exercise 1**

1684 Open a MathPiper session and evaluate the following predicate expressions:

1685 In> 3 = 3

1686 In> 3 = 4

1687 In> 3 < 4

1688 In> 3 != 4

1689 In> -3 < 4

1690 In> 4 >= 4

1691 In> 1/2 < 1/4

1692 In> 15/23 < 122/189

1693 /*In the following two expressions, notice that 1/2 is not considered to be
1694 equal to .5 unless it is converted to a numerical value first.*/

1695 In> 1/2 = .5

1696 In> N(1/2) = .5

1697 **12.3.2 Exercise 2**

1698 Come up with 10 predicate expressions of your own and evaluate them in the
1699 MathPiper console.

1700 **12.4 Making Decisions With The If() Function & Predicate Expressions**

1701 All programming languages have the ability to make decisions and the most
1702 commonly used function for making decisions in MathPiper is the **If()** function.

1703 There are two calling formats for the If() function:

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

1704 The way the first form of the If() function works is that it evaluates the first
1705 expression in its argument list (which is the "**predicate**" expression) and then
1706 looks at the value that is returned. If this value is **True**, the "**then**" expression
1707 that is listed second in the argument list is executed. If the predicate expression
1708 evaluates to **False**, the "**then**" expression is not executed. (Note: any function
1709 that accepts a predicate expression as a parameter can also accept the boolean
1710 values True and False).

1711 The following program uses an If() function to determine if the number in
1712 variable x is greater than 5. If x is greater than 5, the program will echo
1713 "Greater" and then "End of program":

```
1714 1:%mathpiper
1715 2:
1716 3:x := 6;
1717 4:
1718 5:If(x > 5, Echo(x, "is greater than 5.));
1719 6:
1720 7:Echo("End of program.");
1721 8:
1722 9:%/mathpiper
1723 10:
1724 11:    %output,preserve="false"
1725 12:    Result: True
1726 13:
1727 14:    Side Effects:
1728 15:    6 is greater than 5.
1729 16:    End of program.
1730 17:    %/output
```

1731 In this program, x has been set to 6 and therefore the expression $x > 5$ is **True**.
1732 When the **If()** function evaluates the **predicate expression** and determines it is
1733 **True**, it then executes the **first Echo()** function. The **second Echo()** function at
1734 the bottom of the program prints "End of program" regardless of what the If()
1735 function does. (**Note: semicolons cannot be placed after expressions which**
1736 **are in function calls.**)

1737 Here is the same program except that **x** has been set to **4** instead of **6**:

```
1738 1:%mathpiper
1739 2:
1740 3:x := 4;
1741 4:
1742 5:If(x > 5, Echo(x, "is greater than 5.));
1743 6:
1744 7:Echo("End of program.");
1745 8:
1746 9:%/mathpiper
1747 10:
1748 11:    %output,preserve="false"
1749 12:    Result: True
1750 13:
1751 14:    Side Effects:
1752 15:    End of program.
1753 16:    %/output
```

1754 This time the expression $x > 4$ returns a value of **False** which causes the If()
1755 function to not execute the "**then**" expression that was passed to it.

1756 12.4.1 If() Functions Which Include An "Else" Parameter

1757 The second form of the If() function takes a third "**else**" expression which is
1758 executed only if the predicate expression is **False**. This program is similar to the
1759 previous one except an "**else**" expression has been added to it:

```
1760 1:%mathpiper
1761 2:
1762 3:x := 4;
1763 4:
1764 5:If(x > 5, Echo(x, "is greater than 5."), Echo(x, "is NOT greater than 5.));
1765 6:
1766 7:Echo("End of program.");
1767 8:
1768 9:%/mathpiper
1769 10:
1770 11:    %output,preserve="false"
1771 12:    Result: True
1772 13:
1773 14:    Side Effects:
1774 15:    4 is NOT greater than 5.
1775 16:    End of program.
1776 17:    %/output
```

1777 **12.5 Exercises**

1778 **12.5.1 Exercise 1**

1779 Write a program which uses the RandomInteger() function to simulate the
1780 flipping of a coin (Hint: you can use 1 to represent a head and 0 to
1781 represent a tail.). Use predicate expressions, the If() function, and the
1782 Echo() function to print the string "**The coin came up heads.**" or the string
1783 "**The coin came up tails.**", depending on what the simulated coin flip came
1784 up as when the code was executed.

1785 **12.6 The And(), Or(), & Not() Boolean Functions & Infix Notation**

1786 **12.6.1 And()**

1787 Sometimes a programmer needs to check if two or more expressions are all **True**
1788 and one way to do this is with the **And()** function. The And() function has **two**
1789 **calling formats** (or **notations**) and this is the first one:

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

1790 This calling format is able to accept one or more predicate expressions as input.
1791 If **all** of these expressions returns a value of **True**, the And() function will also
1792 return a **True**. However, if **any** of the expressions return a **False**, then the And()
1793 function will return a **False**. This can be seen in the following example:

```
1794 In> And(True, True)  
1795 Result> True
```

```
1796 In> And(True, False)  
1797 Result> False
```

```
1798 In> And(False, True)  
1799 Result> False
```

```
1800 In> And(True, True, True, True)  
1801 Result> True
```

```
1802 In> And(True, True, False, True)  
1803 Result> False
```

1804 The second format (or notation) that can be used to call the And() function is
1805 called **infix** notation:

```
expression1 And expression2
```


1806 With **infix** notation, an expression is placed on both sides of the And() function
1807 name instead of being placed inside of parentheses that are next to it:

1808 In> True And True

1809 Result> True

1810 In> True And False

1811 Result> False

1812 In> False And True

1813 Result> False

1814 Infix notation can only accept **two** expressions at a time, but it is often more
1815 convenient to use than function calling notation. The following program also
1816 demonstrates the infix version of the And() function being used:

```
1817 1:%mathpiper
1818 2:
1819 3:a := 7;
1820 4:b := 9;
1821 5:
1822 6:Echo("1: ", a < 5 And b < 10);
1823 7:Echo("2: ", a > 5 And b > 10);
1824 8:Echo("3: ", a < 5 And b > 10);
1825 9:Echo("4: ", a > 5 And b < 10);
1826 10:
1827 11:If(a > 5 And b < 10, Echo("These expressions are both true.));
1828 12:
1829 13:%/mathpiper
1830 14:
1831 15:    %output,preserve="false"
1832 16:    Result: True
1833 17:
1834 18:    Side Effects:
1835 19:    1: False
1836 20:    2: False
1837 21:    3: False
1838 22:    4: True
1839 23:    These expressions are both true.
1840 23:    %/output
```

1841 12.6.2 Or()

1842 The Or() function is similar to the And() function in that it has both a function
1843 calling format and an infix calling format and it only works with predicate
1844 expressions. However, instead of requiring that all expressions be **True** in order
1845 to return a **True**, Or() will return a **True** if **one or more expressions are True**.

1846 Here is the function calling format for Or():

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

1847 and this example shows Or() being used with function calling format:

1848 In> Or(True, False)

1849 Result> True

1850 In> Or(False, True)

1851 Result> True

1852 In> Or(False, False)

1853 Result> False

1854 In> Or(False, False, False, False)

1855 Result> False

1856 In> Or(False, True, False, False)

1857 Result> True

1858 The infix notation format for Or() is as follows:

```
expression1 Or expression2
```

1859 and this example shows infix notation being used:

1860 In> True Or False

1861 Result> True

1862 In> False Or True

1863 Result> True

1864 In> False Or False

1865 Result> False

1866 The following program also demonstrates the infix version of the Or() function
1867 being used:

```
1868 1:%mathpiper  
1869 2:  
1870 3:a := 7;  
1871 4:b := 9;  
1872 5:  
1873 6:Echo("1: ", a < 5 Or b < 10);  
1874 7:Echo("2: ", a > 5 Or b > 10);
```

```

1875 8:Echo("3: ", a > 5 Or b < 10);
1876 9:Echo("4: ", a < 5 Or b > 10);
1877 10:
1878 11:If(a < 5 Or b < 10,Echo("At least one of these expressions is true.));
1879 12:
1880 13:/mathpiper
1881 14:
1882 15:    %output,preserve="false"
1883 16:    Result: True
1884 17:
1885 18:    Side Effects:
1886 19:    1: True
1887 20:    2: True
1888 21:    3: True
1889 22:    4: False
1890 23:    At least one of these expressions is true.
1891 24:    %/output

```

1892 12.6.3 Not() & Prefix Notation

1893 The **Not()** function works with predicate expressions like the And() and Or()
 1894 functions do, except it can only accept **one** expression as input. The way Not()
 1895 works is that it changes a **True** value to a **False** value and a **False** value to a
 1896 **True** value. Here is the Not()'s function calling format:

```
Not(expression)
```

1897 and this example shows Not() being used with function calling format:

```

1898 In> Not(True)
1899 Result> False

1900 In> Not(False)
1901 Result> True

```

1902 Instead of providing an alternative infix calling format like And() and Or() do,
 1903 Not()'s second calling format uses **prefix** notation:

```
Not expression
```

1904 Prefix notation looks similar to function notation except no parentheses are used:

```

1905 In> Not True
1906 Result> False

```

```
1907 In> Not False
1908 Result> True
```

1909 Finally, here is a program that also uses the prefix version of Not():

```
1910 1:%mathpiper
1911 2:
1912 3:Echo("3 = 3 is ", 3 = 3);
1913 4:
1914 5:Echo("Not 3 = 3 is ", Not 3 = 3);
1915 6:
1916 7:%/mathpiper
1917 8:
1918 9:    %output,preserve="false"
1919 10:    Result: True
1920 11:
1921 12:    Side Effects:
1922 13:    3 = 3 is True
1923 14:    Not 3 = 3 is False
1924 15:    %/output
```

1925 12.7 Exercises

1926 12.7.1 Exercise 1

1927 The following program simulates the rolling of two dice and prints a
1928 message if **both** of the two dice come up less than or equal to 3. Create a
1929 similar program which simulates the flipping of two coins and print the
1930 message "Both coins came up heads." if both coins come up heads.

```
1931 %mathpiper
1932 /*
1933    This program simulates the rolling of two dice and prints a message if
1934    both of the two dice come up less than or equal to 3.
1935 */
```

```
1936 dice1 := RandomInteger(6) + 1;
1937 dice2 := RandomInteger(6) + 1;
```

```
1938 Echo("Dice1: ", dice1, "   Dice2: ", dice2);
1939 NewLine();
```

```
1940 If( dice1 <= 3 And dice2 <= 3, Echo("Both dice came up <= to 3.") );
```

```
1941 %/mathpiper
```

1942 12.7.2 Exercise 2

1943 The following program simulates the rolling of two dice and prints a

```
1944 message if either of the two dice come up less than or equal to 3. Create
1945 a similar program which simulates the flipping of two coins and print the
1946 message "At least one coin came up heads." if at least one coin comes up
1947 heads.

1948 %mathpiper
1949 /*
1950     This program simulates the rolling of two dice and prints a message if
1951     either of the two dice come up less than or equal to 3.
1952 */

1953 dice1 := RandomInteger(6) + 1;
1954 dice2 := RandomInteger(6) + 1;

1955 Echo("Dice1: ", dice1, "   Dice2: ", dice2);
1956 NewLine();

1957 If( dice1 <= 3 Or dice2 <= 3, Echo("At least one die came up <= 3.") );

1958 %/mathpiper
```

13 The While() Looping Function & Bodied Notation

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

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

```
While(predicate)
[
    body_expressions
];
```

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

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

13.1 Printing The Integers From 1 to 10

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

```
1:%mathpiper
2:
3:// This program prints the integers from 1 to 10.
4:
5:
6:/*
7:    Initialize the variable x to 1
8:    outside of the While "loop".
9:*/
10:x := 1;
11:
12:While(x <= 10)
13:[
14:    Echo(x);
```

```
1998 15:
1999 16:     x := x + 1;  //Increment x by 1.
2000 17:];
2001 18:
2002 19: %/mathpiper
2003 20:
2004 21:     %output,preserve="false"
2005 22:     Result: True
2006 23:
2007 24:     Side Effects:
2008 25:         1
2009 26:         2
2010 27:         3
2011 28:         4
2012 29:         5
2013 30:         6
2014 31:         7
2015 32:         8
2016 33:         9
2017 34:        10
2018 35: %/output
```

2019 In this program, a single variable called **x** is created. It is used to tell the Echo()
2020 function which integer to print and it is also used in the predicate expression
2021 that determines if the While() function should continue to **loop** or not.

2022 When the program is executed, 1 is placed into **x** and then the While() function is
2023 called. The predicate expression **x <= 10** becomes **1 <= 10** and, since 1 is
2024 indeed less than or equal to 10, a value of **True** is returned by the predicate
2025 expression.

2026 The While() function sees that the predicate expression returned a **True** and
2027 therefore it executes all of the expressions inside of its **body** from top to bottom.

2028 The Echo() function prints the current contents of x (which is 1) and then the
2029 expression **x := x + 1** is executed.

2030 The expression **x := x + 1** is a standard expression form that is used in many
2031 programming languages. Each time an expression in this form is evaluated, it
2032 **increases the variable it contains by 1**. Another way to describe the effect
2033 this expression has on **x** is to say that it **increments x** by **1**.

2034 In this case **x** contains **1** and, after the expression is evaluated, **x** contains **2**.

2035 After the last expression inside the body of the While() function is executed, the
2036 While() function reevaluates its predicate expression to determine whether it
2037 should continue looping or not. Since **x** is **2** at this point, the predicate
2038 expression returns **True** and the code inside the body of the While() function is
2039 executed again. This loop will be repeated until **x** is incremented to **11** and the
2040 predicate expression returns **False**.

2041 **13.2 Printing The Integers From 1 to 100**

2042 The previous program can be adjusted in a number of ways to achieve different
2043 results. For example, the following program prints the integers from 1 to 100 by
2044 changing the **10** in the predicate expression to **100**. A Write() function is used in
2045 this program so that its output is displayed on the same line until it encounters
2046 the **wrap margin** in MathRider (which can be set in Utilities -> Buffer
2047 Options...).

```
2048 1:%mathpiper
2049 2:
2050 3:// Print the integers from 1 to 100.
2051 4:
2052 5:x := 1;
2053 6:
2054 7:While(x <= 100)
2055 8:[
2056 9:    Write(x, , );
2057 10:
2058 11:    x := x + 1; //Increment x by 1.
2059 12:];
2060 13:
2061 14:%/mathpiper
2062 15:
2063 16:    %output,preserve="false"
2064 17:    Result: True
2065 18:
2066 19:    Side Effects:
2067 20:    1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,
2068    24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,
2069    44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,
2070    64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,
2071    84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100
2072 21:    %/output
```

2073 **13.3 Printing The Odd Integers From 1 To 99**

2074 The following program prints the odd integers from 1 to 99 by changing the
2075 **increment value** in the increment expression from **1** to **2**:

```
2076 1:%mathpiper
2077 2:
2078 3://Print the odd integers from 1 to 99.
2079 4:
2080 5:x := 1;
2081 6:
2082 7:While(x <= 100)
2083 8:[
2084 9:    Write(x, , );
```



```

2085 10:      x := x + 2;  //Increment x by 2.
2086 11:];
2087 12:
2088 13:%/mathpiper
2089 14:
2090 15:      %output,preserve="false"
2091 16:      Result: True
2092 17:
2093 18:      Side Effects:
2094 19:      1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39,41,43,
2095      45,47,49,51,53,55,57,59,61,63,65,67,69,71,73,75,77,79,81,83,
2096      85,87,89,91,93,95,97,99
2097 20:      %/output

```

2098 **13.4 Printing The Integers From 1 To 100 In Reverse Order**

2099 Finally, the following program prints the integers from 1 to 100 in reverse order:

```

2100 1:%mathpiper
2101 2:
2102 3://Print the integers from 1 to 100 in reverse order.
2103 4:
2104 5:x := 100;
2105 6:
2106 7:While(x >= 1)
2107 8:[
2108 9:    Write(x,,);
2109 10:    x := x - 1;  //Decrement x by 1.
2110 11:];
2111 12:
2112 13:%/mathpiper
2113 14:
2114 15:      %output,preserve="false"
2115 16:      Result: True
2116 17:
2117 18:      Side Effects:
2118 19:      100,99,98,97,96,95,94,93,92,91,90,89,88,87,86,85,84,83,82,
2119      81,80,79,78,77,76,75,74,73,72,71,70,69,68,67,66,65,64,63,
2120      62,61,60,59,58,57,56,55,54,53,52,51,50,49,48,47,46,45,44,
2121      43,42,41,40,39,38,37,36,35,34,33,32,31,30,29,28,27,26,25,
2122      24,23,22,21,20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,
2123      3,2,1
2124 20:      %/output

```

2125 In order to achieve the reverse ordering, this program had to initialize **x** to **100**,
 2126 check to see if **x** was **greater than or equal to 1** ($x \geq 1$), and **decrement** **x** by
 2127 **subtracting 1 from it** instead of adding 1 to it.

2128 **13.5 Expressions Inside Of Code Blocks Are Indented**

2129 In the programs in the previous sections which use while loops, notice that the
2130 expressions which are inside of the While() function's code block are **indented**.
2131 These expressions do not need to be indented to execute properly, but doing so
2132 makes the program easier to read.

2133 **13.6 Long-Running Loops, Infinite Loops, & Interrupting Execution**

2134 It is easy to create a loop that will execute a **large number of times**, or even **an**
2135 **infinite number of times**, either on purpose or by mistake. When you execute
2136 a program that contains an **infinite loop**, it will run until you tell MathPiper to
2137 **interrupt** its execution. This is done by opening the MathPiper console and then
2138 pressing the **"Stop"** button which it contains. The Stop button is circular and it
2139 has an X on it. (**Note: currently this button only works if MathPiper is**
2140 **executed inside of a %mathpiper fold.**)

2141 Lets experiment with the **Stop** button by executing a program that contains an
2142 infinite loop and then stopping it:

```
2143 1: %mathpiper
2144 2:
2145 3: //Infinite loop example program.
2146 4:
2147 5: x := 1;
2148 6: While(x < 10)
2149 7: [
2150 8:     answer := x + 1; //Oops, x is not being incremented!.
2151 9: ];
2152 10:
2153 11: %/mathpiper
2154 12:
2155 13: %output,preserve="false"
2156 14:     Processing...
2157 15: %/output
```

2158 Since the contents of x is never changed inside the loop, the expression **x < 10**
2159 always evaluates to **True** which causes the loop to continue looping. Notice that
2160 the %output fold contains the word **"Processing..."** to indicate that the program
2161 is still running the code.

2162 Execute this program now and then interrupt it using the **"Stop"** button. When
2163 the program is interrupted, the %output fold will display the message **"User**
2164 **interrupted calculation"** to indicate that the program was interrupted. After a
2165 program has been interrupted, the program can be edited and then rerun.

2166 **13.7 A Program That Simulates Rolling Two Dice 50 Times**

2167 The following program is larger than the previous programs that have been
2168 discussed in this book, but it is also more interesting and more useful. It uses a
2169 While() loop to simulate the rolling of two dice 50 times and it records how many
2170 times each possible sum has been rolled so that this data can be printed. The
2171 comments in the code explain what each part of the program does. (Remember, if
2172 you copy this program to a MathRider worksheet, you can use **rectangular**
2173 **selection mode** to easily remove the line numbers).

```
2174 1:%mathpiper
2175 2:/*
2176 3:   This program simulates rolling two dice 50 times.
2177 4:*/
2178 5:
2179 6:
2180 7:/*
2181 8:   These variables are used to record how many times
2182 9:   a possible sum of two dice has been rolled. They are
2183 10:  all initialized to 0 before the simulation begins.
2184 11:*/
2185 12:numberOfTwosRolled := 0;
2186 13:numberOfThreesRolled := 0;
2187 14:numberOfFoursRolled := 0;
2188 15:numberOfFivesRolled := 0;
2189 16:numberOfSixesRolled := 0;
2190 17:numberOfSevensRolled := 0;
2191 18:numberOfEightsRolled := 0;
2192 19:numberOfNinesRolled := 0;
2193 20:numberOfTensRolled := 0;
2194 21:numberOfElevensRolled := 0;
2195 22:numberOfTwelvesRolled := 0;
2196 23:
2197 24:
2198 25://This variable keeps track of the number of the current roll.
2199 26:roll := 1;
2200 27:
2201 28:
2202 29:Echo("These are the rolls:");
2203 30:
2204 31:
2205 32:/*
2206 33: The simulation is performed inside of this while loop. The number of
2207 34: times the dice will be rolled can be changed by changing the number 50
2208 35: which is in the While function's predicate expression.
2209 36:*/
2210 37:While(roll <= 50)
2211 38:[
2212 39:   //Roll the dice.
2213 40:   die1 := RandomInteger(6) + 1;
2214 41:   die2 := RandomInteger(6) + 1;
```

```
2215 42:
2216 43:
2217 44: //Calculate the sum of the two dice.
2218 45: rollSum := die1 + die2;
2219 46:
2220 47:
2221 48: /*
2222 49: Print the sum that was rolled. Note: if a large number of rolls
2223 50: is going to be performed (say > 1000), it would be best to comment
2224 51: out this Write() function so that it does not put too much text
2225 52: into the output fold.
2226 53: */
2227 54: Write(rollSum,,);
2228 55:
2229 56:
2230 57: /*
2231 58: These If() functions determine which sum was rolled and then add
2232 59: 1 to the variable which is keeping track of the number of times
2233 60: that sum was rolled.
2234 61: */
2235 62: If(rollSum = 2, numberOfTwosRolled := numberOfTwosRolled + 1);
2236 63: If(rollSum = 3, numberOfThreesRolled := numberOfThreesRolled + 1);
2237 64: If(rollSum = 4, numberOfFoursRolled := numberOfFoursRolled + 1);
2238 65: If(rollSum = 5, numberOfFivesRolled := numberOfFivesRolled + 1);
2239 66: If(rollSum = 6, numberOfSixesRolled := numberOfSixesRolled + 1);
2240 67: If(rollSum = 7, numberOfSevensRolled := numberOfSevensRolled + 1);
2241 68: If(rollSum = 8, numberOfEightsRolled := numberOfEightsRolled + 1);
2242 69: If(rollSum = 9, numberOfNinesRolled := numberOfNinesRolled + 1);
2243 70: If(rollSum = 10, numberOfTensRolled := numberOfTensRolled + 1);
2244 71: If(rollSum = 11, numberOfElevensRolled := numberOfElevensRolled+1);
2245 72: If(rollSum = 12, numberOfTwelvesRolled := numberOfTwelvesRolled+1);
2246 73:
2247 74:
2248 75: //Increment the roll variable to the next roll number.
2249 76: roll := roll + 1;
2250 77:];
2251 78:
2252 79:
2253 80://Print the contents of the sum count variables for visual analysis.
2254 81:NewLine();
2255 82:NewLine();
2256 83:Echo("Number of Twos rolled: ", numberOfTwosRolled);
2257 84:Echo("Number of Threes rolled: ", numberOfThreesRolled);
2258 85:Echo("Number of Fours rolled: ", numberOfFoursRolled);
2259 86:Echo("Number of Fives rolled: ", numberOfFivesRolled);
2260 87:Echo("Number of Sixes rolled: ", numberOfSixesRolled);
2261 88:Echo("Number of Sevens rolled: ", numberOfSevensRolled);
2262 89:Echo("Number of Eights rolled: ", numberOfEightsRolled);
2263 90:Echo("Number of Nines rolled: ", numberOfNinesRolled);
2264 91:Echo("Number of Tens rolled: ", numberOfTensRolled);
2265 92:Echo("Number of Elevens rolled: ", numberOfElevensRolled);
2266 93:Echo("Number of Twelves rolled: ", numberOfTwelvesRolled);
```

```
2267 94:
2268 95:
2269 96: %/mathpiper
2270 97:
2271 98: %output,preserve="false"
2272 99: Result: True
2273 100:
2274 101: Side effects:
2275 102: These are the rolls:
2276 103: 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,
2277 104: 12,7,8,12,6,8,10,10,5,9,8,4,5,3,5,7,7,4,6,7,7,5,8,
2278 105:
2279 106: Number of Twos rolled: 0
2280 107: Number of Threes rolled: 3
2281 108: Number of Fours rolled: 6
2282 109: Number of Fives rolled: 4
2283 110: Number of Sixes rolled: 6
2284 111: Number of Sevens rolled: 13
2285 112: Number of Eights rolled: 6
2286 113: Number of Nines rolled: 3
2287 114: Number of Tens rolled: 2
2288 115: Number of Elevens rolled: 4
2289 116: Number of Twelves rolled: 3
2290 117: . %/output
```

2291 13.8 Exercises

2292 13.8.1 Exercise 1

2293 Create a program which uses a while loop to print the even integers from 2
2294 to 50 inclusive.

2295 13.8.2 Exercise 2

2296 Create a program which prints all the multiples of 5 between 5 and 50
2297 inclusive.

2298 13.8.3 Exercise 3

2299 Create a program which simulates the flipping of a coin 500 times. Print
2300 the number of times the coin came up heads and the number of times it came
2301 up tails after the loop is finished executing.

2302 14 Predicate Functions

2303 A **predicate function** is a function that either returns **True** or **False**. Most
2304 predicate functions in MathPiper have names which begin with "**Is**". For
2305 example, **IsEven()**, **IsOdd()**, **IsInteger()**, etc. The following examples show
2306 some of the predicate functions that are in MathPiper:

```
2307 In> IsEven(4)
2308 Result> True
```

```
2309 In> IsEven(5)
2310 Result> False
```

```
2311 In> IsZero(0)
2312 Result> True
```

```
2313 In> IsZero(1)
2314 Result> False
```

```
2315 In> IsNegativeInteger(-1)
2316 Result> True
```

```
2317 In> IsNegativeInteger(1)
2318 Result> False
```

```
2319 In> IsPrime(7)
2320 Result> True
```

```
2321 In> IsPrime(100)
2322 Result> False
```

2323 There is also an **IsBound()** and an **IsUnbound()** function that can be used to
2324 determine whether or not a value is bound to a given variable:

```
2325 In> a
2326 Result> a
```

```
2327 In> IsBound(a)
2328 Result> False
```

```
2329 In> a := 1
2330 Result> 1
```

```
2331 In> IsBound(a)
2332 Result> True
```

```
2333 In> Clear(a)
2334 Result> True
```

```
2335 In> a
2336 Result> a

2337 In> IsBound(a)
2338 Result> False
```

2339 The complete list of predicate functions is contained in the **User**
2340 **Functions/Predicates** node in the MathPiperDocs plugin.

2341 **14.1 Finding Prime Numbers With A Loop**

2342 Predicate functions are very powerful when they are combined with loops
2343 because they can be used to automatically make numerous checks. The
2344 following program uses a while loop to pass the integers 1 through 20 (one at a
2345 time) to the **IsPrime()** function in order to determine which integers are prime
2346 and which integers are not prime:

```
2347 1:%mathpiper
2348 2:
2349 3://Determine which numbers between 1 and 20 (inclusive) are prime.
2350 4:
2351 5:x := 1;
2352 6:
2353 7:While(x <= 20)
2354 8:[
2355 9:    primeStatus := IsPrime(x);
2356 10:
2357 11:    Echo(x, "is prime: ", primeStatus);
2358 12:
2359 13:    x := x + 1;
2360 14:];
2361 15:
2362 16:%/mathpiper
2363 17:
2364 18:    %output,preserve="false"
2365 19:    Result: True
2366 20:
2367 21:    Side Effects:
2368 22:    1 is prime: False
2369 23:    2 is prime: True
2370 24:    3 is prime: True
2371 25:    4 is prime: False
2372 26:    5 is prime: True
2373 27:    6 is prime: False
2374 28:    7 is prime: True
2375 29:    8 is prime: False
2376 30:    9 is prime: False
2377 31:    10 is prime: False
2378 32:    11 is prime: True
2379 33:    12 is prime: False
```

```

2380 34:      13 is prime: True
2381 35:      14 is prime: False
2382 36:      15 is prime: False
2383 37:      16 is prime: False
2384 38:      17 is prime: True
2385 39:      18 is prime: False
2386 40:      19 is prime: True
2387 41:      20 is prime: False
2388 42:.    %/output

```

2389 This program worked fairly well, but it is limited because it prints a line for each
 2390 prime number and also each non-prime number. This means that if large ranges
 2391 of integers were processed, enormous amounts of output would be produced.
 2392 The following program solves this problem by using an If() function to only print
 2393 a number if it is prime:

```

2394 1:%mathpiper
2395 2:
2396 3://Print the prime numbers between 1 and 50 (inclusive).
2397 4:
2398 5:x := 1;
2399 6:
2400 7:While(x <= 50)
2401 8:[
2402 9:    primeStatus := IsPrime(x);
2403 10:
2404 11:    If(primeStatus = True, Write(x,,) );
2405 12:
2406 13:    x := x + 1;
2407 14:
2408 15:];
2409 16:
2410 17:%/mathpiper
2411 18:
2412 19:    %output,preserve="false"
2413 20:    Result: True
2414 21:
2415 22:    Side Effects:
2416 23:    2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,
2417 24:.    %/output

```

2418 This program is able to process a much larger range of numbers than the
 2419 previous one without having its output fill up the text area. However, the
 2420 program itself can be shortened by moving the **IsPrime()** function **inside** of the
 2421 **If()** function instead of using the **primeStatus** variable to communicate with it:

```

2422 1:%mathpiper
2423 2:
2424 3:/*

```



```
2425 4:      Print the prime numbers between 1 and 50 (inclusive).
2426 5:      This is a shorter version which places the IsPrime() function
2427 6:      inside of the If() function instead of using a variable.
2428 7: */
2429 8:
2430 9: x := 1;
2431 10:
2432 11: While(x <= 50)
2433 12: [
2434 13:
2435 14:     If(IsPrime(x), Write(x, , ) );
2436 15:
2437 16:     x := x + 1;
2438 17:
2439 18: ];
2440 19:
2441 20: %/mathpiper
2442 21:
2443 22:     %output,preserve="false"
2444 23:     Result: True
2445 24:
2446 25:     Side Effects:
2447 26:     2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
2448 27: . %/output
```

2449 14.2 Finding The Length Of A String With The Length() Function

2450 Strings can contain zero or more characters and the **Length()** function can be
2451 used to determine how many characters a string holds:

```
2452 In> s := "Red"
2453 Result> "Red"
```

```
2454 In> Length(s)
2455 Result> 3
```

2456 In this example, the string "Red" is assigned to the variable **s** and then **s** is
2457 passed to the **Length()** function. The **Length()** function returned a **3** which
2458 means the string contained **3 characters**.

2459 The following example shows that strings can also be passed to functions
2460 directly:

```
2461 In> Length("Red")
2462 Result> 3
```

2463 An **empty string** is represented by **two double quote marks with no space in**
2464 **between them**. The **length** of an empty string is **0**:

```
2465 In> Length("")
2466 Result> 0
```

2467 **14.3 Converting Numbers To Strings With The String() Function**

2468 Sometimes it is useful to convert a number to a string so that the individual
2469 digits in the number can be analyzed or manipulated. The following example
2470 shows a **number** being converted to a **string** with the **String()** function so that
2471 its **leftmost** and **rightmost** digits can be placed into **variables**:

```
2472 In> number := 523
2473 Result> 523
```

```
2474 In> stringNumber := String(number)
2475 Result> "523"
```

```
2476 In> leftmostDigit := stringNumber[1]
2477 Result> "5"
```

```
2478 In> rightmostDigit := stringNumber[ Length(stringNumber) ]
2479 Result> "3"
```

2480 Notice that the Length() function is used here to determine which character in
2481 **stringNumber** held the **rightmost** digit.

2482 **14.4 Finding Prime Numbers Which End With 7 (And Multi-line Function Calls)**

2484 Now that we have covered how to turn a number into a string, lets use this
2485 ability inside a loop. The following program finds all the **prime numbers**
2486 between **1** and **500** which have a **7 as their rightmost digit**. There are three
2487 important things which are shown in this program:

2488 1) Function calls **can have their parameters placed on more than one**
2489 **line** if the parameters are too long to fit on a **single line**. In this case, a long
2490 code block is being placed inside of an If() function.

2491 2) Code blocks (which are considered to be compound expressions) **cannot**
2492 **have a semicolon placed after them if they are in a function call**. If a
2493 semicolon is placed after this code block, an error will be produced.

2494 3) If() functions can be placed inside of other If() functions in order to make
2495 more complex decisions. This is referred to as **nesting** functions.

2496 When the program is executed, it finds 24 prime numbers which have 7 as their
2497 rightmost digit:

```

2498 1:%mathpiper
2499 2:
2500 3:/*
2501 4:    Find all the prime numbers between 1 and 500 which have a 7
2502 5:    as their rightmost digit.
2503 6:*/
2504 7:
2505 8:x := 1;
2506 9:
2507 10:While(x <= 500)
2508 11:[
2509 12:    //Notice how function parameters can be put on more than one line.
2510 13:    If(IsPrime(x),
2511 14:        [
2512 15:            stringVersionOfNumber := String(x);
2513 16:
2514 17:            stringLength := Length(stringVersionOfNumber);
2515 18:
2516 19:            //Notice that If() functions can be placed inside of other
2517 20:            // If() functions.
2518 21:            If(stringVersionOfNumber[stringLength] = "7", Write(x,,) );
2519 22:
2520 23:        ] //Notice that semicolons cannot be placed after code blocks
2521 24:        //which are in function calls.
2522 25:
2523 26:    ); //This is the close parentheses for the outer If() function.
2524 27:
2525 28:    x := x + 1;
2526 29:];
2527 30:
2528 31:%/mathpiper
2529 32:
2530 33:    %output,preserve="false"
2531 34:    Result: True
2532 35:
2533 36:    Side Effects:
2534 37:    7,17,37,47,67,97,107,127,137,157,167,197,227,257,277,307,317,
2535 38:    337,347,367,397,457,467,487,
2536 39:    %/output

```

2537 It would be nice if we had the ability to store these numbers someplace so that
 2538 they could be processed further and this is discussed in the next section.

2539 14.5 Exercises

2540 14.5.1 Exercise 1

2541 Write a program which uses a loop to determine how many prime numbers there
 2542 are between 1 and 1000. You do not need to print the numbers themselves,
 2543 just how many there are.

2544 **14.5.2 Exercise 2**

2545 Write a program which uses a loop to print all of the prime numbers between
2546 10 and 99 which contain the digit 3 in either their 1's place, or their
2547 10's place, or both places.

2548 15 Lists: Values That Hold Sequences Of Expressions

2549 The **list** value type is designed to hold expressions in an **ordered collection** or
2550 **sequence**. Lists are very flexible and they are one of the most heavily used
2551 value types in MathPiper. Lists can **hold expressions of any type**, they can be
2552 made to **grow and shrink as needed**, and they can be **nested**. Expressions in a
2553 list can be **accessed by their position** in the list (similar to the way that
2554 characters in a string are accessed) and they can also be **replaced by other**
2555 **expressions**.

2556 One way to create a list is by placing zero or more expressions separated by
2557 commas inside of a **pair of braces {}**. In the following example, a list is created
2558 that contains various expressions and then it is assigned to the variable **x**:

```
2559 In> x := {7,42,"Hello",1/2,var}
```

```
2560 Result> {7,42,"Hello",1/2,var}
```

```
2561 In> x
```

```
2562 Result> {7,42,"Hello",1/2,var}
```

2563 The number of expressions in a list can be determined with the **Length()**
2564 function:

```
2565 In> Length({7,42,"Hello",1/2,var})
```

```
2566 Result> 5
```

2567 A single expression in a list can be accessed by placing a set of **brackets []** to
2568 the right of the variable that is bound to the list and then putting the
2569 expression's position number inside of the brackets (**Note: the first expression**
2570 **in the list is at position 1 counting from the left end of the list**):

```
2571 In> x[1]
```

```
2572 Result> 7
```

```
2573 In> x[2]
```

```
2574 Result> 42
```

```
2575 In> x[3]
```

```
2576 Result> "Hello"
```

```
2577 In> x[4]
```

```
2578 Result> 1/2
```

```
2579 In> x[5]
```

```
2580 Result> var
```

2581 The **1st** and **2nd** expressions in this list are **integers**, the **3rd** expression is a

2582 **string**, the **4th** expression is a **rational number** and the **5th** expression is an
2583 **unbound variable**.

2584 Lists can also hold other lists as shown in the following example:

```
2585 In> x := {20, 30, {31, 32, 33}, 40}
```

```
2586 Result> {20,30,{31,32,33},40}
```

```
2587 In> x[1]
```

```
2588 Result> 20
```

```
2589 In> x[2]
```

```
2590 Result> 30
```

```
2591 In> x[3]
```

```
2592 Result> {31,32,33}
```

```
2593 In> x[4]
```

```
2594 Result> 40
```

```
2595
```

2596 The expression in the **3rd** position in the list is another **list** which contains the
2597 integers **31**, **32**, and **33**.

2598 An expression in this second list can be accessed by two **two sets of brackets**:

```
2599 In> x[3][2]
```

```
2600 Result> 32
```

2601 The **3** inside of the first set of brackets accesses the **3rd** member of the **first** list
2602 and the **2** inside of the second set of brackets accesses the **2nd** member of the
2603 **second** list.

2604 **15.1 Append() & Nondestructive List Operations**

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

2605 The **Append()** function adds an expression to the end of a list:

```
2606 In> testList := {21,22,23}
```

```
2607 Result> {21,22,23}
```

```
2608 In> Append(testList, 24)
```

```
2609 Result> {21,22,23,24}
```

2610 However, instead of changing the **original** list, **Append()** creates a **copy** of the
2611 **original** list and appends the expression to the **copy**. This can be confirmed by
2612 evaluating the variable **testList** after the **Append()** function has been called:

```
2613 In> testList
2614 Result> {21,22,23}
```

2615 Notice that the list that is bound to **testList** was not modified by the **Append()**
2616 function. This is called a **nondestructive list operation** and **most MathPiper**
2617 **functions that manipulate lists do so nondestructively**. To have the new list
2618 bound to the variable that is being used, the following technique can be
2619 employed:

```
2620 In> testList := {21,22,23}
2621 Result> {21,22,23}

2622 In> testList := Append(testList, 24)
2623 Result> {21,22,23,24}
```

```
2624 In> testList
2625 Result> {21,22,23,24}
```

2626 After this code has been executed, the new list has indeed been bound to
2627 **testList** as desired.

2628 There are some functions, such as **DestructiveAppend()**, which **do** change the
2629 original list and most of them begin with the word "Destructive". These are
2630 called "destructive functions" and they are advanced functions which are not
2631 covered in this book.

2632 **15.2 Using While Loops With Lists**

2633 Functions that loop can be used to **select each expression in a list in turn** so
2634 that an operation can be performed on these expressions. The following
2635 program uses a while loop to print each of the expressions in a list:

```
2636 1:%mathpiper
2637 2:
2638 3://Print each number in the list.
2639 4:
2640 5:x := {55,93,40,21,7,24,15,14,82};
2641 6:y := 1;
2642 7:
2643 8:While(y <= Length(x))
2644 9:[
2645 10:     Echo(y, "- ", x[y]);
2646 11:     y := y + 1;
2647 12:];
2648 13:
2649 14:%/mathpiper
2650 15:
2651 16:    %output,preserve="false"
```

```

2652 17:      Result: True
2653 18:
2654 19:      Side Effects:
2655 20:      1 - 55
2656 21:      2 - 93
2657 22:      3 - 40
2658 23:      4 - 21
2659 24:      5 - 7
2660 25:      6 - 24
2661 26:      7 - 15
2662 27:      8 - 14
2663 28:      9 - 82
2664 29:  %/output

```

2665 A **loop** can also be used to search through a list. The following program uses a
 2666 **While()** function and an **If()** function to search through a list to see if it contains
 2667 the number **53**. If 53 is found in the list, a message is printed:

```

2668 1: %mathpiper
2669 2:
2670 3: //Determine if 53 is in the list.
2671 4:
2672 5: testList := {18, 26, 32, 42, 53, 43, 54, 6, 97, 41};
2673 6: index := 1;
2674 7:
2675 8: While(index <= Length(testList))
2676 9: [
2677 10:     If(testList[index] = 53,
2678 11:         Echo("53 was found in the list at position", index));
2679 12:
2680 13:     index := index + 1;
2681 14: ];
2682 15:
2683 16: %/mathpiper
2684 17:
2685 18: %output, preserve="false"
2686 19:      Result: True
2687 20:
2688 21:      Side Effects:
2689 22:      53 was found in the list at position 5
2690 23:  %/output

```

2691 When this program was executed, it determined that **53** was present in the list at
 2692 position **5**.

2693 15.2.1 Using A While Loop And Append() To Place Values In A List

2694 In an earlier section it was mentioned that it would be nice if we could store a set
 2695 of values for later processing and this can be done with a **while loop** and the

2696 **Append()** function. The following program creates an empty list and assigned it
2697 to the variable **primes**. The **while loop** and the **IsPrime()** function is then used
2698 to locate the prime integers between 1 and 50 and the **Append()** function is used
2699 to place them in the list. The last part of the program then prints some
2700 information about the numbers that were placed into the list:

```
2701 1:%mathpiper
2702 2:
2703 3://Place the prime numbers between 1 and 50 (inclusive) into a list.
2704 4:
2705 5://Create an empty list.
2706 6:primes := {};
2707 7:
2708 8:x := 1;
2709 9:
2710 10:While(x <= 50)
2711 11:[
2712 12:    /*
2713 13:        If x is prime, append it to the end of the list and then assign
2714 14:        the new list that is created to the variable 'primes'.
2715 15:    */
2716 16:    If(IsPrime(x), primes := Append(primes, x ) );
2717 17:
2718 18:    x := x + 1;
2719 19:];
2720 20:
2721 21://Print information about the primes that were found.
2722 22:Echo("Primes ", primes);
2723 23:Echo("The number of primes in the list = ", Length(primes) );
2724 24:Echo("The first number in the list = ", primes[1] );
2725 25:
2726 26:%/mathpiper
2727 27:
2728 28:    %output,preserve="false"
2729 29:    Result: True
2730 30:
2731 31:    Side Effects:
2732 32:    Primes {2,3,5,7,11,13,17,19,23,29,31,37,41,43,47}
2733 33:    The number of primes in the list = 15
2734 34:    The first number in the list = 2
2735 35:.    %/output
```

2736 The ability to place values into a list with a loop is very powerful and we will be
2737 using this ability throughout the rest of the book.

2738 **15.3 Exercises**

2739 **15.3.1 Exercise 1**

2740 Create a program that uses a loop and an `IsOdd()` function to analyze the
2741 following list and then print the number of odd numbers it contains.

2742 {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}

2743 **15.3.2 Exercise 2**

2744 Create a program that uses a loop and an `IsNegativeNumber()` function to
2745 copy all of the negative numbers in the following list into a new list.
2746 Use the variable **negativeNumbers** to hold the new list.

2747 {36, -29, -33, -6, 14, 7, -16, -3, -14, 37, -38, -8, -45, -21, -26, 6, 6, 38, -20, 33, 41, -
2748 4, 24, 37, 40, 29}

2749 **15.3.3 Exercise 3**

2750 Create a program that uses a loop to analyze the following list and then
2751 print the following information about it:

- 2752 1) The largest number in the list.
2753 2) The smallest number in the list.
2754 3) The sum of all the numbers in the list.

2755 {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}

2756 **15.4 The `ForEach()` Looping Function**

2757 The **`ForEach()`** function uses a **loop** to index through a list like the `While()`
2758 function does, but it is more flexible and automatic. `ForEach()` also uses bodied
2759 notation like the `While()` function and here is its calling format:

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

2760 **`ForEach()`** selects each expression in a list in turn, assigns it to the passed-in
2761 "variable", and then executes the expressions that are inside of "body".
2762 Therefore, body is **executed once for each expression in the list**.

2763 **15.5 Print All The Values In A List Using A `ForEach()` function**

2764 This example shows how `ForEach()` can be used to print all of the items in a list:

2765 1: %mathpiper
2766 2:

```
2767 3://Print all values in a list.
2768 4:
2769 5:ForEach(x, {50,51,52,53,54,55,56,57,58,59})
2770 6:[
2771 7:     Echo(x);
2772 8:];
2773 9:
2774 10:%mathpiper
2775 11:
2776 12:    %output,preserve="false"
2777 13:    Result: True
2778 14:
2779 15:    Side Effects:
2780 16:        50
2781 17:        51
2782 18:        52
2783 19:        53
2784 20:        54
2785 21:        55
2786 22:        56
2787 23:        57
2788 24:        58
2789 25:        59
2790 26:    %/output
```

2791 15.6 Calculate The Sum Of The Numbers In A List Using ForEach()

2792 In previous examples, counting code in the form $x := x + 1$ was used to count
2793 how many times a while loop was executed. The following program uses a
2794 **ForEach()** function and a line of code similar to this counter to calculate the
2795 **sum of the numbers in a list:**

```
2796 1:%mathpiper
2797 2:
2798 3:/*
2799 4:   This program calculates the sum of the numbers
2800 5:   in a list.
2801 6:*/
2802 7:
2803 8://This variable is used to accumulate the sum.
2804 9:sum := 0;
2805 10:
2806 11:ForEach(x, {1,2,3,4,5,6,7,8,9,10} )
2807 12:[
2808 13:    /*
2809 14:        Add the contents of x to the contents of sum
2810 15:        and place the result back into sum.
2811 16:    */
2812 17:    sum := sum + x;
```

```

2813 18:
2814 19:     //Print the sum as it is being accumulated.
2815 20:     Write(sum,,);
2816 21:];
2817 22:
2818 23:NewLine(); NewLine();
2819 24:
2820 25:Echo("The sum of the numbers in the list = ", sum);
2821 26:
2822 27:~/mathpiper
2823 28:
2824 29:     %output,preserve="false"
2825 30:     Result: True
2826 31:
2827 32:     Side Effects:
2828 33:     1,3,6,10,15,21,28,36,45,55,
2829 34:
2830 35:     The sum of the numbers in the list = 55
2831 36:~ %/output

```

2832 In the above program, the integers **1** through **10** were manually placed into a list
 2833 by typing them individually. This method is limited because only a relatively
 2834 small number of integers can be placed into a list this way. The following section
 2835 discusses an operator which can be used to automatically place a large number
 2836 of integers into a list with very little typing.

2837 **15.7 The .. Range Operator**

```
first .. last
```

2838 A programmer often needs to create a list which contains **consecutive integers**
 2839 and the **.. "range"** operator can be used to do this. The **first** integer in the list is
 2840 placed before the **..** operator and the **last** integer in the list is placed after it
 2841 (**Note: there must be a space immediately to the left of the .. operator**
 2842 **and a space immediately to the right of it or an error will be generated.**).
 2843 Here are some examples:

```

2844 In> 1 .. 10
2845 Result> {1,2,3,4,5,6,7,8,9,10}

2846 In> 10 .. 1
2847 Result> {10,9,8,7,6,5,4,3,2,1}

2848 In> 1 .. 500
2849 Result> {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
2850         21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,
2851         38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,
2852         55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,

```

```
2853         72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,  
2854         89,90,91,92,93,94,95,96,97,98,99,100}
```

```
2855 In> -10 .. 10  
2856 Result> {-10,-9,-8,-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,8,9,10}
```

2857 As these examples show, the `..` operator can generate lists of integers in
2858 ascending order and descending order. It can also generate lists that are very
2859 large and ones that contain negative integers.

2860 Remember, though, if one or both of the spaces around the `..` are omitted, an
2861 error is generated:

```
2862 In> 1..3  
2863 Result>  
2864 Error parsing expression, near token .3.
```

2865 **15.8 Using ForEach() With The Range Operator To Print The Prime** 2866 **Numbers Between 1 And 100**

2867 The following program shows how to use a **ForEach()** function instead of a
2868 **While()** function to print the prime numbers between 1 and 100. Notice that
2869 loops that are implemented with **ForEach()** often require less typing than
2870 their **While()** based equivalents:

```
2871 1:%mathpiper  
2872 2:  
2873 3:/*  
2874 4:   This program prints the prime integers between 1 and 100 using  
2875 5:   a ForEach() function instead of a While() function. Notice that  
2876 6:   the ForEach() version requires less typing than the While()  
2877 7:   version.  
2878 8:*/  
2879 9:  
2880 10:ForEach(x, 1 .. 100)  
2881 11:[  
2882 12:   If(IsPrime(x), Write(x,,) );  
2883 13:];  
2884 14:  
2885 15:%/mathpiper  
2886 16:  
2887 17:   %output,preserve="false"  
2888 18:   Result: True  
2889 19:  
2890 20:   Side Effects:  
2891 21:   2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,  
2892 22:   73,79,83,89,97,  
2893 23:.. %/output
```

2894 15.8.1 Using ForEach() And The Range Operator To Place The Prime 2895 Numbers Between 1 And 50 Into A List

2896 A ForEach() function can also be used to place values in a list, just the the
2897 While() function can:

```

2898 1: %mathpiper
2899 2:
2900 3: /*
2901 4:   Place the prime numbers between 1 and 50 into
2902 5:   a list using a ForEach() function.
2903 6: */
2904 7:
2905 8: //Create a new list.
2906 9: primes := {};
2907 10:
2908 11: ForEach(number, 1 .. 50)
2909 12: [
2910 13:   /*
2911 14:     If number is prime, append it to the end of the list and
2912 15:     then assign the new list that is created to the variable
2913 16:     'primes'.
2914 17:   */
2915 18:   If(IsPrime(number), primes := Append(primes, number) );
2916 19: ];
2917 20:
2918 21: //Print information about the primes that were found.
2919 22: Echo("Primes ", primes);
2920 23: Echo("The number of primes in the list = ", Length(primes) );
2921 24: Echo("The first number in the list = ", primes[1] );
2922 25:
2923 26: %/mathpiper
2924 27:
2925 28:   %output,preserve="false"
2926 29:   Result: True
2927 30:
2928 31:   Side Effects:
2929 32:   Primes {2,3,5,7,11,13,17,19,23,29,31,37,41,43,47}
2930 33:   The number of primes in the list = 15
2931 34:   The first number in the list = 2
2932 35: . %/output

```

2933 As can be seen from the above examples, the **ForEach()** function and the **range**
2934 **operator** can do a significant amount of work with very little typing. You will
2935 discover in the next section that MathPiper has functions which are even more
2936 powerful than these two.

2937 **15.8.2 Exercises**2938 **15.8.3 Exercise 1**

2939 Create a program that uses a **ForEach()** function and an **IsOdd()** function to
2940 analyze the following list and then print the number of odd numbers it
2941 contains.

2942 {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}

2943 **15.8.4 Exercise 2**

2944 Create a program that uses a **ForEach()** function and an **IsNegativeNumber()**
2945 function to copy all of the negative numbers in the following list into a
2946 new list. Use the variable **negativeNumbers** to hold the new list.

2947 {36,-29,-33,-6,14,7,-16,-3,-14,37,-38,-8,-45,-21,-26,6,6,38,-20,33,41,-
2948 4,24,37,40,29}

2949 **15.8.5 Exercise 3**

2950 Create a program that uses a **ForEach()** function to analyze the following
2951 list and then print the following information about it:

- 2952 1) The largest number in the list.
2953 2) The smallest number in the list.
2954 3) The sum of all the numbers in the list.

2955 {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}

2956 **15.8.6 Exercise 4**

2957 Create a program that uses a **while loop** to make a list that contains **1000**
2958 **random integers** between **1** and **100** inclusive. Then, use a **ForEach()**
2959 function to determine how many integers in the list are **prime** and use an
2960 **Echo()** function to print this total.

2961 **16 Functions & Operators Which Loop Internally**

2962 Looping is such a useful capability that MathPiper has many functions which
2963 loop internally. Now that you have some experience with loops, you can use this
2964 experience to help you imagine how these functions use loops to process the
2965 information that is passed to them.

2966 **16.1 Functions & Operators Which Loop Internally To Process Lists**

2967 This section discusses a number of functions that use loops to process lists.

2968 **16.1.1 TableForm()**

```
TableForm(list)
```

2969 The **TableForm()** function prints the contents of a list in the form of a table.
2970 Each member in the list is printed on its own line and this sometimes makes the
2971 contents of the list easier to read:

```
2972 In> testList := {2,4,6,8,10,12,14,16,18,20}  
2973 Result> {2,4,6,8,10,12,14,16,18,20}
```

```
2974 In> TableForm(testList)  
2975 Result> True  
2976 Side Effects>  
2977 2  
2978 4  
2979 6  
2980 8  
2981 10  
2982 12  
2983 14  
2984 16  
2985 18  
2986 20
```

2987 **16.1.2 Contains()**

2988 The **Contains()** function searches a list to determine if it contains a given
2989 expression. If it finds the expression, it returns **True** and if it doesn't find the
2990 expression, it returns **False**. Here is the calling format for Contains():

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


2991 The following code shows Contains() being used to locate a number in a list:

```
2992 In> Contains({50,51,52,53,54,55,56,57,58,59}, 53)
2993 Result> True
```

```
2994 In> Contains({50,51,52,53,54,55,56,57,58,59}, 75)
2995 Result> False
```

2996 The **Not()** function can also be used with predicate functions like Contains() to
2997 change their results to the opposite truth value:

```
2998 In> Not Contains({50,51,52,53,54,55,56,57,58,59}, 75)
2999 Result> True
```

3000 16.1.3 Find()

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

3001 The **Find()** function searches a list for the first occurrence of a given expression.
3002 If the expression is found, the **position of its first occurrence** is returned and
3003 if it is not found, **-1** is returned:

```
3004 In> Find({23, 15, 67, 98, 64}, 15)
3005 Result> 2
```

```
3006 In> Find({23, 15, 67, 98, 64}, 8)
3007 Result> -1
```

3008 16.1.4 Count()

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

3009 **Count()** determines the number of times a given expression occurs in a list:

```
3010 In> testList := {a,b,b,c,c,c,d,d,d,d,e,e,e,e,e}
3011 Result> {a,b,b,c,c,c,d,d,d,d,e,e,e,e,e}
```

```
3012 In> Count(testList, c)
3013 Result> 3
```

```
3014 In> Count(testList, e)
3015 Result> 5
```

```
3016 In> Count(testList, z)
3017 Result> 0
```

3018 **16.1.5 Select()**

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

3019 **Select()** returns a list that contains all the expressions in a list which make a
3020 given predicate function return **True**:

```
3021 In> Select("IsPositiveInteger", {46,87,59,-27,11,86,-21,-58,-86,-52})  
3022 Result> {46,87,59,11,86}
```

3023 In this example, notice that the **name** of the predicate function is passed to
3024 **Select()** in **double quotes**. There are other ways to pass a predicate function to
3025 **Select()** but these are covered in a later section.

3026 Here are some further examples which use the **Select()** function:

```
3027 In> Select("IsOdd", {16,14,82,92,33,74,99,67,65,52})  
3028 Result> {33,99,67,65}
```

```
3029 In> Select("IsEven", {16,14,82,92,33,74,99,67,65,52})  
3030 Result> {16,14,82,92,74,52}
```

```
3031 In> Select("IsPrime", 1 .. 75)  
3032 Result> {2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73}
```

3033 Notice how the third example uses the **..** operator to automatically generate a list
3034 of consecutive integers from 1 to 75 for the **Select()** function to analyze.

3035 **16.1.6 The Nth() Function & The [] Operator**

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

3036 The **Nth()** function simply returns the expression which is at a given position in
3037 a list. This example shows the **third** expression in a list being obtained:

```
3038 In> testList := {a,b,c,d,e,f,g}  
3039 Result> {a,b,c,d,e,f,g}
```

```
3040 In> Nth(testList, 3)  
3041 Result> c
```

3042 As discussed earlier, the **[]** operator can also be used to obtain a single
3043 expression from a list:

```
3044 In> testList[3]
3045 Result> c
```

3046 The **[]** operator can even obtain a single expression directly from a list without
3047 needing to use a variable:

```
3048 In> {a,b,c,d,e,f,g}[3]
3049 Result> c
```

3050 **16.1.7 The : Prepend Operator**

```
expression : list
```

3051 The prepend operator is a colon **:** and it can be used to add an expression to the
3052 beginning of a list:

```
3053 In> testList := {b,c,d}
3054 Result> {b,c,d}

3055 In> testList := a:testList
3056 Result> {a,b,c,d}
```

3057 **16.1.8 Concat()**

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

3058 The Concat() function is short for "concatenate" which means to join together
3059 sequentially. It takes two or more lists and joins them together into a single
3060 larger list:

```
3061 In> Concat({a,b,c}, {1,2,3}, {x,y,z})
3062 Result> {a,b,c,1,2,3,x,y,z}
```

3063 **16.1.9 Insert(), Delete(), & Replace()**

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

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

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

3064 **Insert()** inserts an expression into a list at a given index, **Delete()** deletes an
3065 expression from a list at a given index, and **Replace()** replaces an expression in
3066 a list at a given index with another expression:

```
3067 In> testList := {a,b,c,d,e,f,g}
3068 Result> {a,b,c,d,e,f,g}

3069 In> testList := Insert(testList, 4, 123)
3070 Result> {a,b,c,123,d,e,f,g}

3071 In> testList := Delete(testList, 4)
3072 Result> {a,b,c,d,e,f,g}

3073 In> testList := Replace(testList, 4, xxx)
3074 Result> {a,b,c,xxx,e,f,g}
```

3075 16.1.10 Take()

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

3076 **Take()** obtains a sublist from the **beginning** of a list, the **end** of a list, or the
3077 **middle** of a list. The expressions in the list that are not taken are discarded.

3078 A **positive** integer passed to Take() indicates how many expressions should be
3079 taken from the **beginning** of a list:

```
3080 In> testList := {a,b,c,d,e,f,g}
3081 Result> {a,b,c,d,e,f,g}
```

```
3082 In> Take(testList, 3)
3083 Result> {a,b,c}
```

3084 A **negative** integer passed to Take() indicates how many expressions should be
3085 taken from the **end** of a list:

```
3086 In> Take(testList, -3)
3087 Result> {e,f,g}
```

3088 Finally, if a **two member list** is passed to Take() it indicates the **range** of
3089 expressions that should be taken from the **middle** of a list. The **first** value in the
3090 passed-in list specifies the **beginning** index of the range and the **second** value
3091 specifies its **end**:

```
3092 In> Take(testList, {3,5})
3093 Result> {c,d,e}
```

3094 **16.1.11 Drop()**

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

3095 **Drop()** does the opposite of Take() in that it **drops** expressions from the
3096 **beginning** of a list, the **end** of a list, or the **middle** of a list and **returns a list**
3097 **which contains the remaining expressions.**

3098 A **positive** integer passed to Drop() indicates how many expressions should be
3099 dropped from the **beginning** of a list:

```
3100 In> testList := {a,b,c,d,e,f,g}
3101 Result> {a,b,c,d,e,f,g}
```

```
3102 In> Drop(testList, 3)
3103 Result> {d,e,f,g}
```

3104 A **negative** integer passed to Drop() indicates how many expressions should be
3105 dropped from the **end** of a list:

```
3106 In> Drop(testList, -3)
3107 Result> {a,b,c,d}
```

3108 Finally, if a **two member list** is passed to Drop() it indicates the **range** of
3109 expressions that should be dropped from the **middle** of a list. The **first** value in
3110 the passed-in list specifies the **beginning** index of the range and the **second**
3111 value specifies its **end**:

```
3112 In> Drop(testList, {3,5})
3113 Result> {a,b,f,g}
```

3114 **16.1.12 FillList()**

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

3115 The FillList() function simply creates a list which is of size "length" and fills it
3116 with "length" copies of the given expression:

```
3117 In> FillList(a, 5)
3118 Result> {a,a,a,a,a}
```

```
3119 In> FillList(42,8)
3120 Result> {42,42,42,42,42,42,42,42}
```

3121 **16.1.13 RemoveDuplicates()**

```
RemoveDuplicates(list)
```

3122 **RemoveDuplicates()** removes any duplicate expressions that are contained in
3123 in a list:

```
3124 In> testList := {a,a,b,c,c,b,b,a,b,c,c}
```

```
3125 Result> {a,a,b,c,c,b,b,a,b,c,c}
```

```
3126 In> RemoveDuplicates(testList)
```

```
3127 Result> {a,b,c}
```

3128 **16.1.14 Reverse()**

```
Reverse(list)
```

3129 **Reverse()** reverses the order of the expressions in a list:

```
3130 In> testList := {a,b,c,d,e,f,g,h}
```

```
3131 Result> {a,b,c,d,e,f,g,h}
```

```
3132 In> Reverse(testList)
```

```
3133 Result> {h,g,f,e,d,c,b,a}
```

3134 **16.1.15 Partition()**

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

3135 The **Partition()** function breaks a list into sublists of size "partition_size":

```
3136 In> testList := {a,b,c,d,e,f,g,h}
```

```
3137 Result> {a,b,c,d,e,f,g,h}
```

```
3138 In> Partition(testList, 2)
```

```
3139 Result> {{a,b},{c,d},{e,f},{g,h}}
```

3140 If the partition_size does not divide the length of the list **evenly**, the remaining
3141 elements are discarded:

```
3142 In> Partition(testList, 3)
```

```
3143 Result> {{h,b,c},{d,e,f}}
```

3144 The number of elements that Partition() will discard can be calculated by
3145 dividing the length of a list by the partition size and obtaining the **remainder**:

```
3146 In> Length(testList) % 3  
3147 Result> 2
```

3148 Remember that % is the remainder operator. It divides two integers and returns
3149 their remainder.

3150 16.1.16 Table()

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

3151 The Table() function creates a list of values by doing the following:

- 3152 1) Generating a sequence of values between a "begin_value" and an
3153 "end_value" with each value being incremented by the "step_amount".
- 3154 2) Placing each value in the sequence into the specified "variable", one value
3155 at a time.
- 3156 3) Evaluating the defined "expression" (which contains the defined "variable")
3157 for each value, one at a time.
- 3158 4) Placing the result of each "expression" evaluation into the result list.

3159 This example generates a list which contains the integers 1 through 10:

```
3160 In> Table(x, x, 1, 10, 1)  
3161 Result> {1,2,3,4,5,6,7,8,9,10}
```

3162 Notice that the expression in this example is simply the variable 'x' itself with no
3163 other operations performed on it.

3164 The following example is similar to the previous one except that its expression
3165 multiplies 'x' by 2:

```
3166 In> Table(x*2, x, 1, 10, 1)  
3167 Result> {2,4,6,8,10,12,14,16,18,20}
```

3168 Lists which contain decimal values can also be created by setting the
3169 "step_amount" to a decimal:

```
3170 In> Table(x, x, 0, 1, .1)  
3171 Result> {0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1}
```

3172 **16.2 Functions That Work With Integers**

3173 This section discusses various functions which work with integers. Some of
3174 these functions also work with non-integer values and their use with non-
3175 integers is discussed in other sections.

3176 **16.2.1 RandomIntegerVector()**

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

3177 A vector is a list that does not contain other lists. **RandomIntegerVector()**
3178 creates a list of size "length" that contains random integers that are no lower
3179 than "lowest_possible" and no higher than "highest possible". The following
3180 example creates **10** random integers between **1** and **99** inclusive:

```
3181 In> RandomIntegerVector(10, 1, 99)  
3182 Result> {73,93,80,37,55,93,40,21,7,24}
```

3183 **16.2.2 Max() & Min()**

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

3184 If two values are passed to Max(), it determines which one is larger:

```
3185 In> Max(10, 20)  
3186 Result> 20
```

3187 If a list of values are passed to Max(), it finds the largest value in the list:

```
3188 In> testList := RandomIntegerVector(10, 1, 99)  
3189 Result> {73,93,80,37,55,93,40,21,7,24}
```

```
3190 In> Max(testList)  
3191 Result> 93
```

3192 The **Min()** function is the opposite of the Max() function.

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

3193 If two values are passed to Min(), it determines which one is smaller:

```
3194 In> Min(10, 20)
```


3195 `Result> 10`

3196 If a list of values are passed to `Min()`, it finds the smallest value in the list:

3197 `In> testList := RandomIntegerVector(10, 1, 99)`

3198 `Result> {73,93,80,37,55,93,40,21,7,24}`

3199 `In> Min(testList)`

3200 `Result> 7`

3201 **16.2.3 Div() & Mod()**

`Div(dividend, divisor)`

`Mod(dividend, divisor)`

3202 **Div()** stands for "divide" and determines the whole number of times a divisor
3203 goes into a dividend:

3204 `In> Div(7, 3)`

3205 `Result> 2`

3206 **Mod()** stands for "modulo" and it determines the remainder that results when a
3207 dividend is divided by a divisor:

3208 `In> Mod(7,3)`

3209 `Result> 1`

3210 The remainder/modulo operator `%` can also be used to calculate a remainder:

3211 `In> 7 % 2`

3212 `Result> 1`

3213 **16.2.4 Gcd()**

`Gcd(value1, value2)`

`Gcd(list)`

3214 GCD stands for Greatest Common Divisor and the **Gcd()** function determines the
3215 greatest common divisor of the values that are passed to it.

3216 If two integers are passed to `Gcd()`, it calculates their greatest common divisor:

3217 `In> Gcd(21, 56)`

3218 `Result> 7`

3219 If a list of integers are passed to Gcd(), it finds the greatest common divisor of all
3220 the integers in the list:

```
3221 In> Gcd({9, 66, 123})  
3222 Result> 3
```

3223 16.2.5 Lcm()

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

3224 LCM stands for Least Common Multiple and the **Lcm()** function determines the
3225 least common multiple of the values that are passed to it.

3226 If two integers are passed to Lcm(), it calculates their least common multiple:

```
3227 In> Lcm(14, 8)  
3228 Result> 56
```

3229 If a list of integers are passed to Lcm(), it finds the least common multiple of all
3230 the integers in the list:

```
3231 In> Lcm({3, 7, 9, 11})  
3232 Result> 693
```

3233 16.2.6 Add()

```
Add(value1, value2, ...)  
Add(list)
```

3234 **Add()** can find the sum of two or more values that are passed to it:

```
3235 In> Add(3, 8, 20, 11)  
3236 Result> 42
```

3237 It can also find the sum of a list of values :

```
3238 In> testList := RandomIntegerVector(10, 1, 99)  
3239 Result> {73, 93, 80, 37, 55, 93, 40, 21, 7, 24}
```

```
3240 In> Add(testList)  
3241 Result> 523
```

```
3242 In> testList := 1 .. 10  
3243 Result> {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

```
3244 In> Add(testList)
3245 Result> 55
```

3246 16.2.7 Factorize()

```
Factorize(list)
```

3247 This function has two calling formats, only one of which is discussed here.
3248 **Factorize(list)** multiplies all the expressions in a list together and returns their
3249 product:

```
3250 In> Factorize({1,2,3})
3251 Result> 6
```

3252 16.3 Exercises

3253 16.3.1 Exercise 1

3254 Create a program that uses **RandomIntegerVector()** to create a 100 member
3255 list that contains random integers between 1 and 5 inclusive. Use **Count()**
3256 to determine how many of each digit 1-5 are in the list and then print this
3257 information.

3258 16.3.2 Exercise 2

3259 Create a program that uses **RandomIntegerVector()** to create a 100 member
3260 list that contains random integers between 1 and 50 inclusive and use
3261 **Contains()** to determine if the number 25 is in the list. Print "25 was in
3262 the list." if 25 was found in the list and "25 was not in the list." if it
3263 wasn't found.

3264 16.3.3 Exercise 3

3265 Create a program that uses **RandomIntegerVector()** to create a 100 member
3266 list that contains random integers between 1 and 50 inclusive and use
3267 **Find()** to determine if the number 10 is in the list. Print the position of
3268 10 if it was found in the list and "10 was not in the list." if it wasn't
3269 found.

3270 16.3.4 Exercise 4

3271 Create a program that uses **RandomIntegerVector()** to create a 100 member
3272 list that contains random integers between 0 and 3 inclusive. Use **Select()**
3273 with the **IsNonZeroInteger()** predicate function to obtain all of the nonzero
3274 integers in this list.

3275 **16.3.5 Exercise 5**

3276 Create a program that uses **Table()** to obtain a list which contains the
3277 squares of the integers between 1 and 10 inclusive.

3278 17 Nested Loops

3279 Now that you have seen how to solve problems with single loops, it is time to
3280 discuss what can be done when a loop is placed inside of another loop. A loop
3281 that is placed **inside** of another loop it is called a **nested loop** and this nesting
3282 can be extended to numerous levels if needed. This means that loop 1 can have
3283 loop 2 placed inside of it, loop 2 can have loop 3 placed inside of it, loop 3 can
3284 have loop 4 placed inside of it, and so on.

3285 Nesting loops allows the programmer to accomplish an enormous amount of
3286 work with very little typing.

3287 17.1 Generate All The Combinations That Can Be Entered Into A Two Digit 3288 Wheel Lock Using Two Nested Loops



3289 The following program generates all the combinations that can be entered into a
3290 two digit wheel lock. It uses a nested loop to accomplish this with the "**inside**"
3291 nested loop being used to generate **one's place** digits and the "**outside**" loop
3292 being used to generate **ten's place** digits.

```
3293 1: %mathpiper
3294 2:
3295 3: /*
3296 4:   Generate all the combinations can be entered into a two
3297 5:   digit wheel lock.
3298 6: */
3299 7:
3300 8: combinations := {};
3301 9:
3302 10: ForEach(digit1, 0 .. 9) //This loop is called the "outside" loop.
```

```

3303 11:[
3304 12:     ForEach(digit2, 0 .. 9)//This loop is called the "inside" loop.
3305 13:     [
3306 14:         combinations := Append(combinations, {digit1, digit2});
3307 15:     ];
3308 16:];
3309 17:
3310 18:Echo(TableForm(combinations));
3311 19:
3312 20:%/mathpiper
3313 21:
3314 22:     %output,preserve="false"
3315 23:     Result: True
3316 24:
3317 25:     Side Effects:
3318 26:     {0,0}
3319 27:     {0,1}
3320 28:     {0,2}
3321 29:     {0,3}
3322 30:     {0,4}
3323 31:     {0,5}
3324 32:     {0,6}
3325
3326     . //The middle of the list has not been shown.
3327     .
3328 119:     {9,3}
3329 120:     {9,4}
3330 121:     {9,5}
3331 122:     {9,6}
3332 123:     {9,7}
3333 124:     {9,8}
3334 125:     {9,9}
3335 126:     True
3336 127:.. %/output

```

3337 The relationship between the outside loop and the inside loop is interesting
 3338 because each time the **outside loop cycles once**, the **inside loop cycles 10**
 3339 **times**. Study this program carefully because nested loops can be used to solve a
 3340 wide range of problems and therefore understanding how they work is
 3341 important.

3342 17.2 Exercises

3343 17.2.1 Exercise 1

3344 Create a program that will generate all of the combinations that can be
 3345 entered into a three digit wheel lock. (Hint: a triple nested loop can be
 3346 used to accomplish this.)

3347 18 User Defined Functions

3348 In computer programming, a **function** is a named section of code that can be
3349 **called** from other sections of code. **Values** can be sent to a function for
3350 processing as part of the **call** and a function always returns a value as its result.
3351 A function can also generate side effects when it is called and side effects have
3352 been covered in earlier sections.

3353 The values that are sent to a function when it is called are called **arguments** or
3354 **actual parameters** and a function can accept 0 or more of them. These
3355 arguments are placed within parentheses.

3356 MathPiper has many predefined functions (some of which have been discussed in
3357 previous sections) but users can create their own functions too. The following
3358 program creates a function called **addNums()** which takes two numbers as
3359 arguments, adds them together, and returns their sum back to the calling code
3360 as a result:

```
3361 In> addNums(num1,num2) := num1 + num2
3362 Result> True
```

3363 This line of code defined a new function called **addNums** and specified that it
3364 will accept two values when it is called. The **first** value will be placed into the
3365 variable **num1** and the **second** value will be placed into the variable **num2**.

3366 Variables like num1 and num2 which are used in a function to accept values from
3367 calling code are called **formal parameters**. **Formal parameter variables** are
3368 used inside a function to process the **values/actual parameters/arguments**
3369 that were placed into them by the calling code.

3370 The code on the **right side** of the **assignment operator** is **bound** to the
3371 function name "**addNums**" and it is executed each time **addNums()** is called.
3372 The following example shows the new **addNums()** function being called multiple
3373 times with different values being passed to it:

```
3374 In> addNums(2,3)
3375 Result> 5
```

```
3376 In> addNums(4,5)
3377 Result> 9
```

```
3378 In> addNums(9,1)
3379 Result> 10
```

3380 Notice that, unlike the functions that come with MathPiper, we chose to have this
3381 function's name start with a **lower case letter**. We could have had addNums()
3382 begin with an upper case letter but it is a **convention** in MathPiper for **user**

3383 **defined function names to begin with a lower case letter to distinguish**
3384 **them from the functions that come with MathPiper.**

3385 The values that are returned from user defined functions can also be assigned to
3386 variables. The following example uses a %mathpiper fold to define a function
3387 called **evenIntegers()** and then this function is used in the MathPiper console:

```
3388 1:%mathpiper
3389 2:
3390 3:evenIntegers(endInteger) :=
3391 4:[
3392 5:    resultList := {};
3393 6:    x := 2;
3394 7:
3395 8:    While(x <= endInteger)
3396 9:    [
3397 10:        resultList := Append(resultList, x);
3398 11:        x := x + 2;
3399 12:    ];
3400 13:
3401 14:    resultList;
3402 15:];
3403 16:
3404 17:%/mathpiper
3405 18:
3406 19:    %output,preserve="false"
3407 20:    Result: True
3408 21:    %/output
```

3409 In> a := evenIntegers(10)

3410 Result> {2,4,6,8,10}

3411 In> Length(a)

3412 Result> 5

3413 The function **evenIntegers()** returns a list which contains all the even integers
3414 from 2 up through the value that was passed into it. The fold was first executed
3415 in order to define the **evenIntegers()** function and make it ready for use. The
3416 **evenIntegers()** function was then called from the MathPiper console and 10
3417 was passed to it. After the function was finished executing, it returned a list of
3418 even integers as a result and this result was assigned to the variable 'a'. We then
3419 passed the list that was assigned to 'a' to the **Length()** function in order to
3420 determine its size.

3421 **18.1 Global Variables, Local Variables, & Local()**

3422 The new **evenIntegers()** function seems to work well, but there is a problem.

3423 The variables '**x**' and **resultList** were defined inside the function as **global**
3424 **variables** which means they are accessible from anywhere, including from
3425 within other functions, within other folds (as shown here):

```
3426 1:%mathpiper
3427 2:
3428 3:Echo(x, ",", resultList);
3429 4:
3430 5:%/mathpiper
3431 6:
3432 7:    %output,preserve="false"
3433 8:    Result: True
3434 9:
3435 10:    Side Effects:
3436 11:    12 , {2,4,6,8,10}
3437 12:    %/output
```

3438 and from within the MathPiper console:

```
3439 In> x
3440 Result> 12

3441 In> resultList
3442 Result> {2,4,6,8,10}
```

3443 **Using global variables inside of functions is usually not a good idea**
3444 because code in other functions and folds might already be using (or will use) the
3445 same variable names. Global variables which have the same name are the same
3446 variable. When one section of code changes the value of a given global variable,
3447 the value is changed everywhere that variable is used and this will eventually
3448 cause problems.

3449 In order to prevent errors being caused by global variables having the same
3450 name, a function named **Local()** can be called inside of a function to define what
3451 are called **local variables**. A **local variable** is only accessible inside the
3452 function it has been defined in, even if it has the same name as a global variable.
3453 The following example shows a second version of the **evenIntegers()** function
3454 which uses **Local()** to make '**x**' and **resultList** local variables:

```
3455 1:%mathpiper
3456 2:
3457 3:/*
3458 4: This version of evenIntegers() uses Local() to make
3459 5: x and resultList local variables
3460 6:*/
3461 7:
3462 8:evenIntegers(endInteger) :=
3463 9:[
```

```
3464 10:    Local(x,resultList);
3465 11:
3466 12:    resultList := {};
3467 13:    x := 2;
3468 14:
3469 15:    While(x <= endInteger)
3470 16:    [
3471 17:        resultList := Append(resultList, x);
3472 18:        x := x + 2;
3473 19:    ];
3474 20:
3475 21:    resultList;
3476 22:];
3477 23:
3478 24: %/mathpiper
3479 25:
3480 26:    %output,preserve="false"
3481 27:    Result: True
3482 28:    %/output
```

3483 We can verify that '**x**' and **resultList** are now local variables by first clearing
3484 them, calling **evenIntegers()**, and then seeing what '**x**' and **resultList** contain:

```
3485 In> Clear(x, resultList)
3486 Result> True

3487 In> evenIntegers(10)
3488 Result> {2,4,6,8,10}

3489 In> x
3490 Result> x

3491 In> resultList
3492 Result> resultList
```

3493 18.2 Exercises

3494 18.2.1 Exercise 1

3495 Create a function called **tenOddIntegers()** which returns a list which
3496 contains 10 random odd integers between 1 and 99 inclusive.

3497 18.2.2 Exercise 2

3498 Create a function called **convertStringToList(string)** which takes a string
3499 as a parameter and returns a list which contains all of the characters in
3500 the string. Here is an example of how the function should work:

```
3501 In> convertStringToList("Hello friend!")
3502 Result> {"H","e","l","l","o"," ","f","r","i","e","n","d","!"}
```

```
3503 In> convertStringToList("Computer Algebra System")
3504 Result> {"C","o","m","p","u","t","e","r"," ","A","l","g","e","b","r","a","
3505 ","s","y","s","t","e","m"}
```

3506 **THE INFORMATION BELOW THIS LINE IS STILL IN DEVELOPMENT**

3507 19 Models

3508 Up to this point in the book, a significant number of software-based tools have
3509 been discussed and in this section you will start to learn how to use these tools to
3510 do useful work. Doing useful work is the main reason that computers were
3511 invented. However, before one can understand how to useful work with a
3512 computer, one must first understand what a **model** is.

3513 One example of a model that most people are familiar with is a scaled-down
3514 plastic model car. When a scaled-down version of an object is made it means
3515 that a smaller copy of the object is created, with each of the dimensions of all of
3516 its parts being shrunk by the same amount. For example, if a scaled-down car
3517 was 50 times smaller than a given full-size car, then all of the parts in the scaled-
3518 down car would be 50 times smaller than their analogous parts in the full-size
3519 car.

3520 Of course, the model car does not contain small working copies of all of the parts
3521 of a real car because it would be very difficult to create small working copies of
3522 all of the parts in a real car. It probably could be done, but it would be very
3523 expensive. This is why models are usually used to **represent** objects instead of
3524 either scaled or unscaled exact **copies** of the objects. A **model** is a simplified
3525 **representation** of an object that only represents **some** of its attributes.
3526 Examples of typical object attributes include weight, height, strength, and color.

3527 The attributes that are selected for representation are chosen for a given
3528 purpose. The more attributes that are represented in the model, the more
3529 expensive the model is to make. Therefore, only those attributes that are
3530 absolutely needed to achieve a given purpose are usually represented in a model.
3531 The process of selecting only some of an object's attributes when developing a
3532 model of it is called **abstraction**.

3533 As an example, suppose we wanted to build a garage that could hold two cars
3534 along with a workbench, a set of storage shelves, and a riding lawn mower.
3535 Assuming that the garage will have an adequate ceiling height, and that we do
3536 not want to build the garage any larger than it needs to be for our stated
3537 purpose, how could an adequate length and width be determined for the garage?

3538 One strategy for determining the size of the garage is to build perhaps 10
3539 garages of various sizes in a large field. When the garages are finished, take two
3540 cars to the field along with a workbench, a set of storage shelves, and a riding
3541 lawn mower. Then, place these items into each garage in turn to see which is the
3542 smallest one that these items will fit into without being too cramped. The test
3543 garages in the field can then be discarded and a garage which is the same size as
3544 the one that was chosen could be built at the desired location."

3545 If you are thinking "Thats ridiculous, 11 garages would need to be built using
3546 this strategy instead of just one." you are correct. A way to solve the problem
3547 less expensively is by using a model of the garage and models of the items that

3548 will be placed inside it.

3549 Since we only want to determine the dimensions of the garage's floor, we can
3550 make a scaled down model of just its floor, maybe using a piece of paper. Each of
3551 the items that will be placed into the garage could also be represented by scaled-
3552 down pieces of paper. Then, the pieces of paper that represent the items can be
3553 placed on top of the the large piece of paper that represents the floor and these
3554 smaller pieces of paper can be moved around to see how they fit. If the items
3555 are too cramped, a larger piece of paper can be cut to represent the floor and, if
3556 the items have too much room, a smaller piece of paper for the floor can be cut.

3557 When a good fit is found, the length and width of the piece of paper that
3558 represents the floor can be measured and then these measurements can be
3559 scaled up to the units used for the full-size garage. With this method, only a few
3560 pieces of paper are needed to solve the problem instead of 10 full-size garages
3561 that will later be discarded.

3562 What makes these pieces of paper models of the full-size objects they represent
3563 and not exact scaled-down copies of them is that the only attributes of the full-
3564 sized objects that were represented by pieces of paper were the object's length
3565 and width. The process of placing only some of an object's attributes into a
3566 model instead of all of them is called abstraction.

3567 ***19.1 Placing Models Into A Computer***

3568 Now that we have discussed what a model is you may find it interesting to know
3569 that the reason one of the first modern programmable digital computers was
3570 invented was to model the paths of artillery projectiles. Its name was ENIAC and
3571 it was was invented in the 1940s. This shows that the process of modeling with
3572 computers is one of the fundamental reasons that people use computers and
3573 using computers as modeling tools has continued to the present day.

3574 As discussed in the previous section, simple tools like paper, scissors, and a
3575 pencil can be used to create models, but tools like these are very limited in their
3576 modeling abilities

3577 "I can see how paper can be used to model things...

