

# **Introduction To Programming With MathPiper And MathPiperIDE**

**by Ted Kosan**

Copyright © 2017 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	Website And Support Email List.....	8
1.3	Recommended Weekly Sequence When Teaching A Class With This Book. .	8
2	Introduction.....	9
2.1	What Is A Mathematics Computing Environment?.....	9
2.2	What Is MathPiperIDE?.....	10
2.3	What Inspired The Creation Of MathPiperIDE?.....	11
3	Downloading, Installing, And Executing MathPiperIDE.....	12
3.1	MathPiperIDE's Directory Structure.....	12
4	The Graphical User Interface.....	13
4.1	Buffers And Text Areas.....	13
4.2	The Gutter.....	13
4.3	Menus.....	13
4.3.1	File.....	14
4.3.2	Edit.....	14
4.3.3	Search.....	14
4.3.4	Markers, Folding, and View.....	15
4.3.5	Utilities.....	15
4.3.6	Macros.....	15
4.3.7	Plugins.....	15
4.3.8	Help.....	15
4.4	The Toolbar.....	15
4.4.1	Undo And Redo.....	16
5	Using MathPiperIDE As A Programmer's Text Editor.....	17
5.1	Creating, Opening, Saving, And Closing Text Files.....	17
5.2	Editing Files.....	17
5.3	File Modes.....	18
5.4	Learning How To Type Properly Is An Excellent Investment Of Your Time	18
5.5	Exercises.....	18
5.5.1	Exercise 1.....	18
6	MathPiper: A Computer Algebra System For Beginners.....	19
6.1	Numeric Vs. Symbolic Computations.....	19
6.2	Using The MathPiper Console As A Numeric (Scientific) Calculator.....	20
6.2.1	Using Procedures.....	22
6.2.1.1	The Sqrt() Square Root Procedure.....	22
6.2.1.2	The Even?() Procedure.....	23
6.2.2	Accessing Previous Input And Results.....	23

6.3 Saving And Restoring A Console Session.....	23
6.3.1 Syntax Errors.....	24
6.4 Using The MathPiper Console As A Symbolic Calculator.....	24
6.4.1 Variables And The Variable State.....	25
6.4.1.1 The Global Variable State.....	26
6.4.1.2 Evaluating An Unassigned Variable Throws An Exception.....	27
6.4.1.3 Constants.....	27
6.4.1.4 Calculating With Constants.....	29
6.4.1.5 Variable And Constant Names Are Case Sensitive.....	30
6.4.1.6 Using More Than One Variable.....	30
6.5 Exercises.....	31
6.5.1 Exercise 1.....	31
6.5.2 Exercise 2.....	31
6.5.3 Exercise 3.....	31
6.5.4 Exercise 4.....	32
7 The MathPiper Documentation Plugin.....	33
7.1 Procedure List.....	33
7.2 Mini Web Browser Interface.....	33
7.3 Exercises.....	34
7.3.1 Exercise 1.....	34
7.3.2 Exercise 2.....	34
8 MathPiperIDE Worksheet Files.....	35
8.1 Code Folds And Source Code.....	35
8.1.1 The title Attribute.....	36
8.2 Automatically Inserting Folds & Removing Unpreserved Folds.....	36
8.3 Placing Text Outside Of A Fold.....	37
8.4 Rectangular Selection Mode And Text Area Splitting.....	37
8.4.1 Rectangular Selection Mode.....	37
8.4.2 Text area splitting.....	38
8.4.3 Exercises.....	38
8.4.3.1 Exercise 1.....	39
8.4.3.2 Exercise 2.....	39
8.4.3.3 Exercise 3.....	39
8.4.3.4 Exercise 4.....	39
9 MathPiper Programming Fundamentals.....	40
9.1 Values, Literals, And Expressions.....	40
9.2 Operators.....	40
9.3 Operator Precedence.....	41
9.4 Changing The Order Of Operations In An Expression.....	44
9.5 Procedures & Procedure Names.....	45
9.6 Procedures That Produce Side Effects.....	45

9.6.1 Printing Related Procedures: Echo(), Write(), And Newline().....	46
9.6.1.1 The Echo() Procedure.....	46
9.6.1.2 Echo Procedures Are Useful For "Debugging" Programs.....	48
9.6.1.3 Write().....	48
9.6.1.4 NewLine().....	49
9.7 Expressions Are Separated By Semicolons.....	49
9.7.1 Placing More Than One Expression On A Line In A Fold.....	50
9.7.2 Placing Consecutive Expressions Into A Code Sequence.....	51
9.7.2.1 Automatic Bracket, Parentheses, And Brace Match Indicating.....	52
9.8 Strings.....	53
9.8.1 The MathPiper Console and MathPiper Folds Can Access The Same Variables.....	53
9.8.2 Using Strings To Make Echo's Output Easier To Read.....	53
9.8.2.1 Combining Strings With The + Operator.....	54
9.8.2.2 WriteString().....	54
9.8.2.3 Nl().....	55
9.8.2.4 Space().....	55
9.8.3 Accessing The Individual Letters/Characters In A String.....	55
9.8.3.1 Indexing Before The Beginning Of A String Or Past The End Of A String.....	56
9.9 Comments.....	56
9.10 How To Tell If MathPiper Has Crashed And What To Do If It Has.....	58
9.11 Exercises.....	58
9.11.1 Exercise 1.....	58
9.11.2 Exercise 2.....	59
9.11.3 Exercise 3.....	59
9.11.4 Exercise 4.....	59
9.11.5 Exercise 5.....	59
9.11.6 Exercise 6.....	60
10 Lists.....	61
10.1 Append!().....	62
11 Random Integer Values.....	63
11.1 Obtaining Random Integers With The RandomInteger() Procedure.....	63
11.2 Simulating The Rolling Of Dice.....	64
11.3 Exercises.....	65
11.3.1 Exercise 1.....	65
12 Making Decisions.....	66
12.1 Relational Operators.....	66
12.2 Predicate Expressions.....	69
12.3 Exercises.....	69
12.3.1 Exercise 1.....	69

12.3.2 Exercise 2.....	70
12.4 Making Decisions With The If() Procedure & Predicate Expressions.....	70
12.4.1 One If() Procedure Used With One Else Operator.....	73
12.5 The &?,  ?, And !? Boolean Operators.....	74
12.5.1 The &? "And" Operator.....	74
12.5.2 The  ? "Or" Operator.....	75
12.5.3 The !? "Not" Operator.....	76
12.6 Exercises.....	77
12.6.1 Exercise 1.....	77
12.6.2 Exercise 2.....	78
12.6.3 Exercise 3.....	78
13 The While() And Until() Looping Procedures.....	79
13.1 The While() Looping Procedure.....	79
13.1.1 Printing The Integers From 1 to 10.....	79
13.1.2 Placing The Integers From 1 to 50 In A List.....	81
13.1.3 Printing The Odd Integers From 1 To 99.....	81
13.1.4 Placing The Integers From 1 To 100 In Reverse Order Into A List....	82
13.2 The Until() Looping Procedure.....	83
13.2.1 Printing The Integers From 1 to 10.....	83
13.3 Expressions Inside Of Code Sequences Are Indented.....	84
13.4 Long-Running Loops, Infinite Loops, & Interrupting Execution.....	84
13.5 A Program That Simulates Rolling Two Dice 50 Times.....	85
13.6 Exercises.....	87
13.6.1 Exercise 1.....	88
13.6.2 Exercise 2.....	88
13.6.3 Exercise 3.....	88
14 Predicate Procedures.....	89
14.1 Finding Prime Numbers With A Loop.....	90
14.2 Finding The Length Of A String With The Length() Procedure.....	91
14.3 Converting Numbers To Strings With The ToString() Procedure.....	92
14.4 Finding Prime Numbers that End With 7 (And Multi-line Procedure Calls) .....	92
14.5 Exercises.....	94
14.5.1 Exercise 1.....	94
14.5.2 Exercise 2.....	94
15 More Applications Of Using While Loops With Lists.....	95
15.1 Adding 1 To Each Element In A List.....	95
15.2 Determining If A Number Is In A List.....	95
15.3 Finding The Sum Of The Integers In A List Using A While Loop.....	96
15.4 Exercises.....	97
15.4.1 Exercise 1.....	97

15.4.2 Exercise 2.....	97
15.4.3 Exercise 3.....	98
15.5 The ForEach() Looping Procedure.....	99
15.6 Print All The Values In A List Using A ForEach() procedure.....	99
15.7 Calculate The Sum Of The Numbers In A List Using ForEach().....	99
15.8 The .. Range Operator.....	100
15.9 Using ForEach() With The Range Operator To Print The Prime Numbers Between 1 And 100.....	101
15.9.1 Using ForEach() And The Range Operator To Place The Prime Numbers Between 1 And 50 Into A List.....	102
15.9.2 Exercises.....	103
15.9.3 Exercise 1.....	104
15.9.4 Exercise 2.....	104
15.9.5 Exercise 3.....	104
15.9.6 Exercise 4.....	104
16 Procedures & Operators That Loop Internally.....	105
16.1 Procedures & Operators That Loop Internally To Process Lists.....	105
16.1.1 TableForm().....	105
16.1.2 Contains?().....	105
16.1.3 Find().....	106
16.1.4 Count().....	106
16.1.5 Select().....	107
16.1.6 The Nth() Procedure & The [] Operator.....	107
16.1.7 Concat().....	108
16.1.8 Insert(), Delete(), & Replace().....	108
16.1.9 Take().....	109
16.1.10 Drop().....	109
16.1.11 FillList().....	110
16.1.12 RemoveDuplicates().....	110
16.1.13 Reverse().....	111
16.1.14 Partition().....	111
16.1.15 BuildList().....	112
16.1.16 Sort().....	112
16.2 Procedures That Work With Integers.....	113
16.2.1 RandomIntegerList().....	113
16.2.2 Maximum() & Minimum().....	113
16.2.3 Quotient() & Modulo().....	114
16.2.4 Gcd().....	114
16.2.5 Lcm().....	115
16.2.6 Sum().....	115
16.2.7 Product().....	116
16.3 Exercises.....	116

16.3.1 Exercise 1.....	116
16.3.2 Exercise 2.....	117
16.3.3 Exercise 3.....	117
16.3.4 Exercise 4.....	117
16.3.5 Exercise 5.....	117
17 Nested Loops.....	118
17.1 Generate All The Combinations That Can Be Entered Into A Two Digit Wheel Lock Using A Nested Loop.....	118
17.2 Exercises.....	119
17.2.1 Exercise 1.....	120
18 User Defined Procedures.....	121
18.1 Global Variables, Local Variables, & Local().....	123
18.2 Exercises.....	125
18.2.1 Exercise 1.....	125
18.2.2 Exercise 2.....	125
19 Miscellaneous topics.....	127
19.1 Incrementing And Decrementing Variables With The ++ And -- Operators .....	127
19.1.1 Incrementing Variables With The ++ Operator.....	127
19.1.2 Decrementing Variables With The -- Operator.....	128
19.1.3 The For() Looping Procedure.....	129
19.1.4 The Break() Procedure.....	130
19.1.5 The Continue() Procedure.....	130
19.1.6 The Repeat() Looping Procedure.....	131
19.1.7 The EchoTime() Procedure.....	133
19.2 Exercises.....	135
19.2.1 Exercise 1.....	135
19.2.2 Exercise 2.....	135
19.2.3 Exercise 3.....	136
19.2.4 Exercise 4.....	136
19.2.5 Exercise 5.....	136

# 1 Preface

## 2 1.1 Dedication

3 This book is dedicated to Steve Yegge and his blog entries "Math Every Day"  
4 (<http://steve.yegge.googlepages.com/math-every-day>) and "Math For  
5 Programmers" ([http://steve-yegge.blogspot.com/2006/03/math-for-](http://steve-yegge.blogspot.com/2006/03/math-for-programmers.html)  
6 [programmers.html](http://steve-yegge.blogspot.com/2006/03/math-for-programmers.html)).

## 7 1.2 Website And Support Email List

8 The website for MathPiper and MathPiperIDE is <http://mathpiper.org>.

9 The support email list for this book is called **mathpiper-**  
10 **user@googlegroups.com** and you can subscribe to it at  
11 <http://groups.google.com/group/mathpiper-user>

## 12 1.3 Recommended Weekly Sequence When Teaching A Class With This 13 Book

- 14 • Week 1: Sections 1 - 7.
- 15 • Week 2: Sections 8 - 9.
- 16 • Week 3: Sections 10 - 13.
- 17 • Week 4: Sections 14 - 15.
- 18 • Week 5: Sections 16 - 19.
- 19 • Week 6: Exam



## 20 **2 Introduction**

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

### 30 **2.1 What Is A Mathematics Computing Environment?**

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

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

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

47 [http://en.wikipedia.org/wiki/Comparison\\_of\\_computer\\_algebra\\_systems](http://en.wikipedia.org/wiki/Comparison_of_computer_algebra_systems)

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

53 As an example of the difference between traditional mathematics form and text  
54 form, here is a formula that is displayed in traditional form:

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

55 and here is the same formula in text form:

56 
$$a = x^2 + 4*h*x + 3/7$$

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

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

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

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

## 78 **2.2 What Is MathPiperIDE?**

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

84 MathPiperIDE uses MathPiper as its default computer algebra system, BeanShell  
85 as its main scripting language, jEdit as its development environment, and Java as  
86 its overall implementation language. One way to determine a person's  
87 MathPiperIDE expertise is by their knowledge of these components. (see Table 1)

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

*Table 1: MathPiperIDE user experience levels.*

88 This book is for MathPiperIDE and programming beginners. This book will teach  
 89 you enough programming to begin solving problems with MathPiperIDE using  
 90 the MathPiper programming language. It will help you to become a  
 91 MathPiperIDE Novice, but you will need to learn MathPiper from books that are  
 92 dedicated to it before you can become a MathPiperIDE Expert.

93 The MathPiperIDE project website (<http://mathpiper.org>) contains more  
 94 information about MathPiperIDE along with other MathPiperIDE resources.

## 95 **2.3 What Inspired The Creation Of MathPiperIDE?**

96 One of the main inspirations for MathPiper is Steve Yegge's thoughts on learning  
 97 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) The right way to learn math is breadth-first, not depth-first. You need to  
 101 survey the space, learn the names of things, figure out what's what.

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

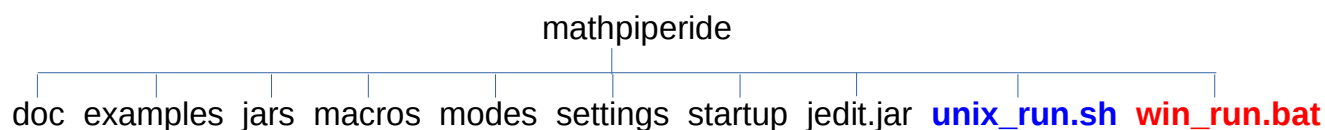
### 103 **3 Downloading, Installing, And Executing MathPiperIDE**

104 Instructions for downloading and installing MathPiperIDE are on the download  
105 page of the MathPiper website (<http://mathpiper.org>).

#### 106 **3.1 MathPiperIDE's Directory Structure**

107 The top level of MathPiperIDE's directory structure is shown in Illustration 1:

*Illustration 1: MathPiperIDE's Directory Structure*



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

109 **doc** - Contains MathPiperIDE's documentation files.

110 **examples** - Contains various example programs, some of which are pre-opened  
111 when MathPiperIDE is first executed.

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

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

114 **modes** - Contains files that tell MathPiperIDE how to do syntax highlighting for  
115 various file types.

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

117 **startup** - Contains startup scripts that are executed each time MathPiperIDE  
118 launches.

119 **jedit.jar** - Holds the core jEdit application that MathPiperIDE builds upon.

120 **unix\_run.sh** - The script used to execute MathPiperIDE on Unix systems.

121 **win\_run.bat** - The batch file used to execute MathPiperIDE on Windows  
122 systems.

## 123 4 The Graphical User Interface

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

132 Most software is developed with a programmer's text editor (or environments  
133 that contain one) and so learning how to use a programmer's text editor is one of  
134 the many skills that MathPiperIDE provides that can be used in other areas. The  
135 MathPiperIDE series of books are designed so that these capabilities are  
136 revealed to the reader over time.

137 In the following sections, the main parts of MathPiperIDE's graphical user  
138 interface are briefly covered. Some of these parts are covered in more depth  
139 later in the book and some are covered in other books.

140 **As you read through the following sections, I encourage you to explore**  
141 **each part of MathPiperIDE that is being discussed using your own copy**  
142 **of MathPiperIDE.**

### 143 4.1 Buffers And Text Areas

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

### 150 4.2 The Gutter

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

### 154 4.3 Menus

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

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

### 161 4.3.1 File

162 The File menu contains actions that are typically found in normal text editors and  
163 word processors. The actions to create new files, save files, and open existing  
164 files are all present along with variations on these actions.

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

### 167 4.3.2 Edit

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

### 173 4.3.3 Search

174 The actions in the Search menu are used heavily, even by beginners. A good way  
175 to get your mind around the search actions is to open the Search dialog window  
176 by selecting the **Find...** action, which is the first actions in the Search menu. A  
177 **Search And Replace** dialog window will then appear that contains access to  
178 most of the search actions.

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

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

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

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

196 contents of the **Search for** text area and replace them with the contents of the  
197 **Replace with** text area.

#### 198 4.3.4 Markers, Folding, and View

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

#### 200 4.3.5 Utilities

201 The utilities menu contains a significant number of actions, some that are useful  
202 to beginners and others that are meant for experts. The two actions that are  
203 most useful to beginners are the **Buffer Options** actions and the **Global**  
204 **Options** actions. The **Buffer Options** actions allows the currently selected  
205 buffer to be customized and the **Global Options** actions brings up a rich dialog  
206 window that allows numerous aspects of the MathPiperIDE application to be  
207 configured.

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

#### 209 4.3.6 Macros

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

#### 211 4.3.7 Plugins

212 Plugins are component-like pieces of software that are designed to provide an  
213 application with extended capabilities and they are similar in concept to physical  
214 world components. The tabs on the right side of the application that are labeled  
215 "JFreeChart", "MathPiper", "MathPiperDocs", etc. are all plugins and they can be  
216 **opened** and **closed** by clicking on their **tabs**. **Feel free to close any of these**  
217 **plugins, which may be opened if you are not currently using them.**  
218 MathPiperIDE pPlugins are covered in more depth in a later section.

#### 219 4.3.8 Help

220 The most important action in the **Help** menu is the **MathPiperIDE Help** action.  
221 This action brings up a dialog window with contains documentation for the core  
222 MathPiperIDE application along with documentation for each installed plugin.

#### 223 4.4 The Toolbar

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

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

#### 231 **4.4.1 Undo And Redo**

232 The **Undo** button on the toolbar is able to undo any text was entered since the  
233 current session of MathPiperIDE was launched. This is very handy for undoing  
234 mistakes or getting back text that was deleted. The **Redo** button can be used if  
235 you have selected Undo too many times and you need to "undo" one ore more  
236 Undo operations.



## 237 **5 Using MathPiperIDE As A Programmer's Text Editor**

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

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

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

### 252 **5.1 Creating, Opening, Saving, And Closing Text Files**

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

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

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

### 265 **5.2 Editing Files**

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

## 271 **5.3 File Modes**

272 Text file names are suppose to have a file extension that indicates what type of  
273 file it is. For example, test.**txt** is a generic text file, test.**bat** is a Windows batch  
274 file, and test.**sh** is a Unix/Linux shell script (**unfortunately, Windows is usually**  
275 **configured to hide file extensions, but viewing a file's properties by right-clicking**  
276 **on it will show this information.**).

277 MathPiperIDE uses a file's extension type to place its text area into a customized  
278 **mode** that highlights various parts of its contents. For example, MathPiperIDE  
279 worksheet files have a **.mpws** extension and MathPiperIDE knows what colors to  
280 highlight the various parts of a .mpws file in.

## 281 **5.4 Learning How To Type Properly Is An Excellent Investment Of Your** 282 **Time**

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

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

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

## 293 **5.5 Exercises**

### 294 **5.5.1 Exercise 1**

295 Create a text file called "**my\_text\_file.txt**" and place a few sentences in  
296 it. Save the text file somewhere on your hard drive then close it. Now,  
297 open the text file again using **File->Open** and verify that what you typed is  
298 still in the file.

## 299 **6 MathPiper: A Computer Algebra System For Beginners**

300 Computer algebra systems are extremely powerful and very useful for solving  
301 STEM-related problems. In fact, one of the reasons for creating MathPiperIDE  
302 was to provide a vehicle for delivering a computer algebra system to as many  
303 people as possible. If you like using a scientific calculator, you should love using  
304 a computer algebra system!

305 At this point you may be asking yourself "if computer algebra systems are so  
306 wonderful, why aren't more people using them?" One reason is that most  
307 computer algebra systems are complex and difficult to learn. Another reason is  
308 that proprietary systems are very expensive and therefore beyond the reach of  
309 most people. Luckily, there are some open source computer algebra systems  
310 that are powerful enough to keep most people engaged for years, and yet simple  
311 enough that even a beginner can start using them. MathPiper, which is based on  
312 a CAS called Yacas, is one of these simpler computer algebra systems and it is  
313 the computer algebra system that is included by default with MathPiperIDE.

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

### 317 **6.1 Numeric Vs. Symbolic Computations**

318 A Computer Algebra System (CAS) is software that is capable of performing both  
319 **numeric** and **symbolic** computations. **Numeric** computations are performed  
320 exclusively with numerals and these are the type of computations that are  
321 performed by typical hand-held calculators.

322 **Symbolic** computations (which also called algebraic computations) relate "...to  
323 the use of machines, such as computers, to manipulate mathematical equations  
324 and expressions in symbolic form, as opposed to manipulating the  
325 approximations of specific numerical quantities represented by those symbols."  
326 ([http://en.wikipedia.org/wiki/Symbolic\\_mathematics](http://en.wikipedia.org/wiki/Symbolic_mathematics)).

327 Since most people who read this document will probably be familiar with  
328 performing numeric calculations as done on a scientific calculator, the next  
329 section shows how to use MathPiper as a scientific calculator. The section after  
330 that then shows how to use MathPiper as a symbolic calculator. Both sections  
331 use the console interface to MathPiper. In MathPiperIDE, a console interface to  
332 any plugin or application is a text-only **shell** or **command line** interface to it.  
333 This means that you type on the keyboard to send information to the console and  
334 it prints text to send you information.

## 335 **6.2 Using The MathPiper Console As A Numeric (Scientific) Calculator**

336 Open the MathPiperConsole plugin by selecting the **MathPiperConsole** tab in  
337 the lower left part of the MathPiperIDE application. The MathPiper **console**  
338 interface is a text area that is inside this plugin. The size of the console text area  
339 can be changed by dragging on the dotted lines that are at the top side and right  
340 side of the console window.

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

343 MathPiper version "xxx".

344 In>

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

347 In> 2+2

348 Result: 4

349 In>

350 When **<enter>** was pressed,  $2 + 2$  was read into MathPiper for **evaluation** and  
351 **Result:** was printed followed by the result **4**. The numeral 4 is the **value** that  
352 was returned by **evaluating**  $2 + 2$ . Another input prompt was then displayed so  
353 that further input could be entered. This **input, evaluation, output** process  
354 will continue as long as the console is running and it is sometimes called a **Read,**  
355 **Eval, Print Loop** or **REPL**. In further examples, the last **In>** prompt will not be  
356 shown to save space.

357 Previous input can be automatically entered to the right of an In> prompt by  
358 placing the cursor to the right of the prompt, pressing the **<ctrl>** key, and then  
359 pressing the up and down arrow keys.

360 In addition to addition, MathPiper can also do subtraction, multiplication,  
361 exponents, and division:

362 In> 5-2

363 Result: 3

364 In> 3\*4

365 Result: 12

366 In> 2^3

367 Result: 8

368 In> 12/6

369 Result: 2

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

375 MathPiper can also work with decimal numbers:

```
376 In> .5+1.2  
377 Result: 1.7
```

```
378 In> 3.7-2.6  
379 Result: 1.1
```

```
380 In> 2.2*3.9  
381 Result: 8.58
```

```
382 In> 2.2^3  
383 Result: 10.648
```

```
384 In> 1/2  
385 Result: 1/2
```

386 In the last example, MathPiper returned the fraction unevaluated. This  
387 sometimes happens due to MathPiper's symbolic nature, but a result in **numeric**  
388 **form** can be obtained by using the **NM() procedure** (which is discussed in the  
389 next section):

```
390 In> NM(1/2)  
391 Result: 0.5
```

392 As can be seen here, when a result is given in numeric form, it means that it is  
393 given as a **decimal number**. A numeric result could also be obtained by using a  
394 decimal point either after the 1 or the 2 (or both of them):

```
395 In> 1./2  
396 Result: 0.5
```

```
397 In> 1/2.  
398 Result: 0.5
```

```
399 In> 1./2.  
400 Result: 0.5
```

401 When one or more decimal numbers are used in a calculation, MathPiper will  
402 usually return a numeric result.

## 403 6.2.1 Using Procedures

404 **NM()** is an example of a **procedure**. A procedure can be thought of as a "black  
405 box" that accepts input, processes the input, and returns a result. Each  
406 procedure has a name and in this case, the name of the procedure is **NM**, which  
407 stands for "**Numeric Mode**". To the right of a procedure's name there is always  
408 a **set of parentheses**, and information that is sent to the procedure is placed  
409 inside of them. The purpose of the **NM()** procedure is to make sure that the  
410 information that is sent to it is processed numerically instead of symbolically.  
411 Procedures are used by **evaluating** them, and this happens when <enter> is  
412 pressed. Another name for evaluating a procedure is **calling** it.

### 413 6.2.1.1 The Sqrt() Square Root Procedure

414 The following example show the **NM()** procedure being used with the square  
415 root procedure **Sqrt()**:

```
416 In> Sqrt(9)
417 Result: 3
```

```
418 In> Sqrt(8)
419 Result: Sqrt(8)
```

```
420 In> NM(Sqrt(8))
421 Result: 2.828427125
```

422 Notice that Sqrt(9) returned 3 as expected but Sqrt(8) returned Sqrt(8). We  
423 needed to use the NM() procedure to force the square root procedure to return a  
424 numeric result. The reason that Sqrt(8) does not appear to have done anything  
425 is because computer algebra systems like to work with expressions that are as  
426 exact as possible. In this case the **symbolic** value Sqrt(8) represents the number  
427 that is the square root of 8 more accurately than any decimal number can.

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

430 2.828427125

431 2.82842712474619

432 2.82842712474619009760337744842

433 2.8284271247461900976033774484193961571393437507539

434 Whenever MathPiper returns a symbolic result and a numeric result is desired,  
435 simply use the NM() procedure to obtain one. The ability to work with symbolic  
436 values are one of the things that make computer algebra systems so powerful,  
437 and they are discussed in more depth in later sections.

### 438 6.2.1.2 The Even?() Procedure

439 An example of a simple procedure is **Even?()**. The **Even?()** procedure takes a  
440 number as input and returns **True** if the number is even and **False** if it is not  
441 even:

442 In> Even?(4)

443 Result: True

444 In> Even?(5)

445 Result: False

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

### 451 6.2.2 Accessing Previous Input And Results

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

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

459 Finally, the MathPiperConsole associates the most recent computation result  
460 with the number sign (**#**) character. If you want to use the most recent result in  
461 a new calculation, access it with this character:

462 In> 5\*8

463 Result: 40

464 In> #

465 Result: 40

466 In> # \* 2 **(Note: there needs to be a space between the # and \* characters.)**

467 Result: 80

### 468 6.3 Saving And Restoring A Console Session

469 If you need to save or open the contents of a console session, you can use the  
470 "File" menu that is present in the upper left corner of the MathPiperConsole  
471 window.

### 472 6.3.1 Syntax Errors

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

478 In> 12 \ 6

479 Exception: Error encountered during parsing: Error parsing expression near  
480 token \*\*\*( \ )\*\*\*. Starting at index 3

481 To fix this problem, change the \ to a /, and reevaluate the expression.

482 This section provided a short introduction to using MathPiper as a numeric  
483 calculator. The next section contains a short introduction to using MathPiper as a  
484 symbolic calculator.

### 485 6.4 Using The MathPiper Console As A Symbolic Calculator

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

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

491 In> 1/2 + 1/3  
492 Result: 5/6

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

497 In> #  
498 Result: 5/6

499 Let's say that you would like to have MathPiper determine the numerator of this  
500 result. This can be done by using (or **calling**) the **Numerator()** procedure:

501 In> Numerator(#)  
502 Result: 5



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

#### 6.4.1 Variables And The Variable State

What would be nice is if MathPiper provided a way to assign **results** (which are also called **values**) to symbols that we choose, instead of ones that it chooses. Fortunately, this is exactly what it does! **Names** that can be associated with values are called **variables**. Variable names must start with an upper or lower case letter and be followed by zero or more upper case letters, lower case letters, or numbers. Examples of variable names include:

**a, b, x, y, answer, totalAmount, and index.**

Even though variable names can start with an upper case letter, by convention all variables should begin with a lower case letter. If the name is composed of more than one word, the first letter of each word after the first word should be capitalized as shown in these examples:

**numberOfDoors, seatsInRoom6, and averageTemperature.**

**Note: the underscore (\_) character cannot be used in variable names. One or more underscore characters in a name identify it as a constant. A constant is a name that always evaluates to itself, and it is discussed shortly.**

The process of associating a value with a variable is called **assigning** the value to the variable, and this consists of placing the name of a **variable** you would like to create on the **left** side of the **assignment operator** (`:=`) and an **expression** on the **right** side of this operator. This expression is evaluated, and the value it returns is **assigned** to the variable. For example, the following code assigns the value 5 to the variable 'a':

```
In> a := 5
Result: 5
```

The assignment operator (`:=`) is read as "**becomes**", and therefore the above expression reads " **'a' becomes 5**".

If the variable 'a' is evaluated by itself, it returns the value that is currently assigned to it:

```
In> a
Result: 5
```

537 The assignment operator ( $:=$ ) is meant to look like an arrow that points from  
538 right to left like  $\leftarrow$  in order to emphasize the right-to-left assignment of variables.

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

540 In> a := (1/2 + 1/3)

541 Result: 5/6

542 In> a

543 Result: 5/6

544 In> Numerator(a)

545 Result: 5

546 In> Denominator(a)

547 Result: 6

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

550 **most recently assigned to it (in this case  $\frac{5}{6}$ ) was returned.** This value will

551 stay assigned to the variable 'a' as long as MathPiper is running, unless 'a' is  
552 unassigned with the **Unassign()** procedure, or 'a' has another value assigned to  
553 it. This is why we were able to determine both the numerator and the  
554 denominator of the rational number assigned to 'a' using two procedures in turn.

555 **(Note: there can be no spaces between the : and the = in the := operator)**

#### 556 6.4.1.1 The Global Variable State

557 The **global variable state** is the list of all of the global variables that are  
558 currently assigned, along with the values that have been assigned to them. A  
559 global variable is a variable that is accessible by all the code in the system. The  
560 other main kind of variable is a local variable. Local variables (which are covered  
561 in a later section) are accessible to limited sections of code. All variables that we  
562 will be using in the MathPiper console are global variables.

563 The **State()** procedure can be used to obtain a copy of the global variable state:

564 In> a := 1

565 Result: 1

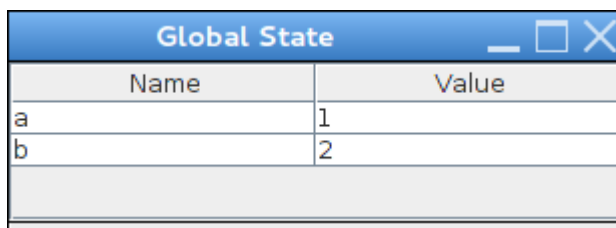
566 In> b := 2

567 Result: 2

568 In> State()

569 Result: [a:1,b:2]

570 The **State** button in the console can also be used to view the global state. When  
571 this button is pressed, a window is shown that contains the global variable state:



Name	Value
a	1
b	2

572 It is a good idea to keep a current variable state window open while  
573 programming because it makes it easier to see what effects the code is  
574 producing.

#### 575 **6.4.1.2 Evaluating An Unassigned Variable Throws An Exception**

576 If an unassigned variable is evaluated, an exception is thrown:

577 In> Unassign(a)

578 Result: True

579 In> a

580 Result: Exception

581 Exception: The variable <a> does not have a value assigned to it.

582 The Unassign() procedure unassigns a variable, and it returns the value **True** as  
583 a result to indicate that the variable that was sent to it was successfully  
584 **unassigned**. Many procedures return either return **True** or **False** to indicate  
585 whether or not the operation they performed succeeded. True and False are  
586 constants, and constants are discussed in the next section.

#### 587 **6.4.1.3 Constants**

588 A **constant** is a name that evaluates to itself. The following is list of some  
589 constants that are predefined in MathPiper:

- 590 • True
- 591 • False
- 592 • Infinity
- 593 • Undefined
- 594 • All
- 595 • None

596 The constant **Infinity** evaluates to itself:

```
597 In> Infinity
598 Result: Infinity
```

599 If an attempt is made to assign a value to a constant, an exception is thrown:

```
600 In> Infinity := 3
601 Result: Exception
602 Exception: <Infinity> is a constant, and values cannot be assigned to
603 constants.
```

604 As mentioned earlier, some procedures return a predefined constant as a value.  
605 For example, the Assigned?() procedure returns **True** if a variable is assigned,  
606 and it returns **False** if it is unassigned:

```
607 In> a := 1
608 Result: 1
```

```
609 In> a
610 Result: 1
```

```
611 In> Assigned?(a)
612 Result: True
```

613 **All** currently assigned variables can be unassigned by passing the **constant 'All'**  
614 **to Unassign**:

```
615 In> b := 2
616 Result: 2
```

```
617 In> State()
618 Result: [a:1,b:2]
```

```
619 In> Unassign(All)
620 Result: True
```

```
621 In> State()
622 Result: []
```

623 One way to indicate that a name is a constant is to use an underscore character  
624 ( \_ ) as the first letter in the name:

625 **\_x, \_y, \_heavy**

626 Constants that start with an underscore evaluate to themselves:

```
627 In> _x
628 Result: _x
```

629 and values cannot be assigned to these constants either:

```
630 In> _x := 3
631 Result: Exception
632 <_x> is a constant, and values cannot be assigned to constants.
```

633 Numbers are also constants because they evaluate to themselves:

```
634 In> 3
635 Result: 3
```

#### 636 6.4.1.4 Calculating With Constants

637 **Constants** may not appear to be very useful, but they provide the flexibility  
638 needed for computer algebra systems to perform symbolic calculations. In order  
639 to demonstrate this flexibility, let's first factor some numbers using the **Factor()**  
640 procedure:

```
641 In> Factor(8)
642 Result: 2^3
```

```
643 In> Factor(14)
644 Result: 2*7
```

```
645 In> Factor(2343)
646 Result: 3*11*71
```

647 Now let's factor an expression that contains the constant '\_x':

```
648 In> Factor(_x^2 + 24*_x + 80)
649 Result: (_x+4)*(_x+20)
```

```
650 In> Expand(#)
651 Result: _x^2+24*_x+80
```

652 **Factor()** uses the rules of algebra to **manipulate** the algebraic expression that  
653 is sent to it into factored form. The **Expand()** procedure was then able to take  
654 the factored expression  $(\_x+4)(\_x+20)$  and manipulate it until it was expanded.  
655 One way to remember what the procedures **Factor()** and **Expand()** do is to look  
656 at the second letters of their names. The 'a' in **Factor** can be thought of as  
657 **adding** parentheses to an expression, and the 'x' in **Expand** can be thought of  
658 **xing** out or removing parentheses from an expression.

#### 659 **6.4.1.5 Variable And Constant Names Are Case Sensitive**

660 MathPiper variable and constant names are **case sensitive**. This means  
661 MathPiper takes into account the **case** of each letter in a variable name when it  
662 is deciding if two or more variable names are the same variable or not. For  
663 example, the variable name **Box** and the variable name **box** are not the same  
664 variable because the first variable name starts with an upper case 'B' and the  
665 second variable name starts with a lower case 'b':

```
666 In> Box := 1
667 Result: 1
```

```
668 In> box := 2
669 Result: 2
```

```
670 In> Box
671 Result: 1
```

```
672 In> box
673 Result: 2
```

#### 674 **6.4.1.6 Using More Than One Variable**

675 Programs are able to have more than one variable. The following example shows  
676 three variables being used:

```
677 In> a := 2
678 Result: 2
```

```
679 In> b := 3
680 Result: 3
```

```
681 In> a + b
682 Result: 5
```

```
683 In> answer := (a + b)
684 Result: 5
```

```
685 In> answer
686 Result: 5
```

687 The part of an expression that is on the **right side** of an assignment operator is  
688 always evaluated first, and the result value is then assigned to the variable that  
689 is on the **left side** of the operator.

690 Now that you have seen how to use the MathPiper console as both a **symbolic**  
691 and a **numeric** calculator, our next step is to take a closer look at the procedures  
692 that are included with MathPiper. As you will soon discover, MathPiper contains

693 numerous procedures that deal with a wide range of mathematics.

## 694 **6.5 Exercises**

695 Use the MathPiper console that is at the bottom of the MathPiperIDE application  
696 to complete the following exercises.

### 697 **6.5.1 Exercise 1**

698 Answer each one of the following questions:

699 a) What is the purpose of the NM() procedure?

700 b) What is a variable?

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

702 d) What is the difference between an assigned variable and an unassigned  
703 variable?

704 e) What happens if you evaluate an unassigned variable?

705 f) How can a value be assigned to a variable?

706 g) How can a variable be unassigned?

707 h) What does the # character do?

### 708 **6.5.2 Exercise 2**

709 Perform the following calculation:

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

### 710 **6.5.3 Exercise 3**

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

713 In> **answer** := (1/5 + 7/4 + 15/16)

714 b) Use the Numerator() procedure to calculate the numerator of **answer**.

715 c) Use the Denominator() procedure to calculate the denominator of **answer**.

716 d) Use the NM() procedure to calculate the numeric value of **answer**.

717 e) Use the Unassign() procedure to unassign the variable **answer** and verify

718 that **answer** is unassigned by using the State() procedure and by using the  
719 Global State window.

#### 720 6.5.4 Exercise 4

721 Assign  $\frac{1}{4}$  to variable **x**,  $\frac{3}{8}$  to variable **y**, and  $\frac{7}{16}$  to variable **z** using the  
722 := operator (**remember, there is no space between the : and the =**). Then  
723 perform the following calculations:

724 a)

725 In> x

726 b)

727 In> y

728 c)

729 In> z

730 d)

731 In> x + y

732 e)

733 In> x + z

734 f)

735 In> x + y + z



## 736 **7 The MathPiper Documentation Plugin**

737 MathPiper has a significant amount of reference documentation written for it  
738 and this documentation has been placed into a plugin called **MathPiperDocs** in  
739 order to make it easier to navigate. The MathPiperDocs plugin is available in a  
740 tab called "MathPiperDocs", which is near the right side of the MathPiperIDE  
741 application. Click on this tab to open the plugin and click on it again to close it.

742 The left side of the MathPiperDocs window contains the names of all the  
743 procedures that come with MathPiper and the right side of the window contains  
744 a mini-browser that can be used to navigate the documentation.

### 745 **7.1 Procedure List**

746 MathPiper's procedures are divided into two main categories called **user**  
747 procedures and **programmer procedures**. In general, the **user procedures**  
748 are used for solving problems in the MathPiper console or with short programs  
749 and the **programmer procedures** are used for longer programs. However,  
750 users will often use some of the programmer procedures and programmers will  
751 use the user procedures as needed.

752 Both the user and programmer procedure names have been placed into a "tree"  
753 on the left side of the MathPiperDocs window to allow for easy navigation. The  
754 branches of the procedure tree can be opened and closed by clicking on the  
755 small "circle with a line attached to it" symbol, which is to the left of each  
756 branch. Both the user and programmer branches have the procedures they  
757 contain organized into categories and the **top category in each branch** lists all  
758 the procedures in the branch in **alphabetical order** for quick access. Clicking  
759 on a procedure will bring up documentation about it in the browser window and  
760 selecting the **Collapse** button at the top of the plugin will collapse the tree.

761 **Don't be intimidated by the large number of categories and procedures**  
762 **that are in the procedure tree!** Most MathPiperIDE beginners will not know  
763 what most of them mean, and some will not know what any of them mean. Part  
764 of the benefit MathPiperIDE provides is exposing the user to the existence of  
765 these categories and procedures. The more you use MathPiperIDE, the more  
766 you will learn about these categories and procedures and someday you may even  
767 get to the point where you understand all of them. This book is designed to show  
768 beginners how to begin using these procedures using a gentle step-by-step  
769 approach.

### 770 **7.2 Mini Web Browser Interface**

771 MathPiper's reference documentation is in HTML (or web page) format and so  
772 the right side of the plugin contains a mini web browser that can be used to  
773 navigate through these pages. The browser's **home page** contains links to the

774 main parts of the MathPiper documentation. As links are selected, the **Back** and  
775 **Forward** buttons in the upper right corner of the plugin allow the user to move  
776 backward and forward through previously visited pages and the **Home** button  
777 navigates back to the home page.

778 The procedure names in the procedure tree all point to sections in the HTML  
779 documentation so the user can access procedure information either by  
780 navigating to it with the browser or jumping directly to it with the procedure  
781 tree.

## 782 **7.3 Exercises**

### 783 **7.3.1 Exercise 1**

784 Locate the `NM()`, `Even?()`, `Odd?()`, `Unassign()`, `Assigned?()`, `Numerator()`,  
785 `Denominator()`, and `State()` procedures in the **All Procedures** section of the  
786 MathPiperDocs plugin, and read the information that is available on them.  
787 List the **one line descriptions** that are at the top of the documentation for  
788 each of these procedures.

### 789 **7.3.2 Exercise 2**

790 Locate the `NM()`, `Even?()`, `Odd?()`, `Unassign()`, `Assigned?()`, `Numerator()`,  
791 `Denominator()`, and `State()` procedures in the **Mathematical Procedures**  
792 section or the **Programming Procedures** section of the MathPiperDocs plugin  
793 and list which **category** each procedure is contained in. **Don't** include the  
794 **Alphabetical** or **Built In** categories in your search. For example, the `NM()`  
795 procedure is in the **Numbers (Operations)** category.

## 8 MathPiperIDE Worksheet Files

While MathPiperIDE's ability to execute code inside a console provides a significant amount of power to the user, most of MathPiperIDE's power is derived from **worksheets**. MathPiperIDE worksheets are text files that have a **.mpws** extension and are which are able to execute multiple types of code in a single text area. The **worksheet\_demo\_1.mpws** file (which is preloaded in the MathPiperIDE environment when it is first launched) demonstrates how a worksheet is able to execute multiple types of code in what are called **code folds**. (**Note: a new .mpws file needs to be saved immediately after it is created, because MathPiperIDE will not recognize it as a MathPiper worksheet until it has been saved.**)

### 8.1 Code Folds And Source Code

A code fold is a named section inside a MathPiperIDE worksheet that contains source code which can be executed by placing the cursor inside of it and pressing **<shift><Enter>**. One or more expressions that are typed into a code fold are called a **computer program**, and these expressions are the program's **source code**. A fold always begins with a **start tag**, which starts with a percent symbol "%" followed by the **name of the fold type** (like this: **%<foldtype>**). The end of a fold is marked by an **end tag**, which looks like **%/<foldtype>**. The only difference between a fold's start tag and its end tag is that the end tag has a slash "/" after the "%".

For example, here is a MathPiper fold that will print the result of **2 + 3** to the MathPiper console (**Note: the semicolon ";" that is at the end of the line of code is required.**):

```
%mathpiper
2 + 3;
%/mathpiper
```

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

```
%mathpiper
  2 + 3;
%/mathpiper
```

```
830     %output,preserve="false"  
831     Result: 5  
832 .    %/output
```

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

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

### 840 8.1.1 The title Attribute

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

```
844 %mathpiper,title="Add two numbers together."  
845 2 + 3;  
846 %/mathpiper
```

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

## 851 8.2 Automatically Inserting Folds & Removing Unpreserved Folds

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

```
853 %mathpiper  
854 %/mathpiper
```

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

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

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

### 865 **8.3 Placing Text Outside Of A Fold**

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

867 Text can be placed above folds like this.

```
868 text text text text  
869 text text text text
```

```
870 %mathpiper,title="Fold 1"
```

```
871 2 + 3;
```

```
872 %/mathpiper
```

873 Text can be placed between folds like this.

```
874 text text text text  
875 text text text text
```

```
876 %mathpiper,title="Fold 2"
```

```
877 3 + 4;
```

```
878 %/mathpiper
```

879 Text can be placed after folds like this.

```
880 text text text text  
881 text text text text
```

882 Placing text above a fold is useful for describing what is being done inside the  
883 fold.

## 884 **8.4 Rectangular Selection Mode And Text Area Splitting**

### 885 **8.4.1 Rectangular Selection Mode**

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

1) Type three or four lines of text into a text area.

2) Hold down the **<Alt>** key (or the **<control>** key on Macintosh computers) then slowly press the **backslash key** (\) a few times. The bottom of the MathPiperIDE window contains a text field that MathPiperIDE uses to communicate information to the user. As **<Alt>\** is repeatedly pressed, messages are displayed that 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.**

4) Holding down the **<CTRL>** key (or the **<command>** key on Macintosh computers) in regular selection mode will temporarily place the system into rectangular selection mode.

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

## 8.4.2 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. MathPiperIDE has this ability and it is called **splitting**.

If you look just to the right of the toolbar there is an icon that looks like a blank window, an icon to the right of it that looks like a window that was split horizontally, and an icon to the right of the horizontal one that is split vertically. If you let your mouse hover over these icons, a short description will be displayed for each of them.

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.

## 8.4.3 Exercises

A MathPiperIDE worksheet file called **"intro\_book\_examples\_1.mpws"** is included in the mathpiperide/examples directory and **it is opened by default** when the software is first launched after it is downloaded. It contains a number of %mathpiper folds that contain code examples from the previous sections of this book. Notice that all of the lines of code have a semicolon (;) placed after them. The reason this is needed is explained in a later section.

Download this worksheet file to your computer from the section on this website that contains the highest revision number and then open it in MathPiperIDE.

926 Then, use the worksheet to do the following exercises.

927 **8.4.3.1 Exercise 1**

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

930 **8.4.3.2 Exercise 2**

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

933 **8.4.3.3 Exercise 3**

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

935 **8.4.3.4 Exercise 4**

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

939 a)  $2*7 + 3$

940 b)  $18/3$

941 c)  $234238342 + 2038408203$

942 d)  $324802984 * 2308098234$

943 e) Factor the result that was calculated in d).

## 944 **9 MathPiper Programming Fundamentals**

945 The MathPiper language consists of **expressions** and an expression consists of  
946 one or more **symbols** that represent **values**, **operators**, **variables**, and  
947 **procedures**. In this section expressions are explained along with the values,  
948 operators, variables, and procedures they consist of.

### 949 **9.1 Values, Literals, And Expressions**

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

952 `3`

953 represents the number three, the value:

954 `0.5`

955 represents the number one half, and the value:

956 `"Mathematics is powerful!"`

957 is a "string" of characters that represents an English sentence (strings are  
958 covered in a later section).

959 A **literal** is any notation in computer source code that represents a value. Any  
960 number that is present in the source code of a program is a literal. For example,  
961 the 3 above is an integer number literal, and the number 0.5 is a real number  
962 literal. Additional literals will be discussed in later sections.

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

966 `3`

967 `2 + 3`

968 `5 + 6*21/7 - 2^3`

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

972 `In> 2 + 3`

973 `Result: 5`

### 974 **9.2 Operators**

975 In the above expressions, the characters `+`, `-`, `*`, `/`, `^` are called **operators** and



976 their purpose is to tell MathPiper what **operations** to perform on the **values** in  
977 an **expression**. For example, in the expression **2 + 3**, the **addition** operator **+**  
978 tells MathPiper to add the integer **3** to the integer **2** and return the result.

979 The **subtraction** operator is **-**, the **multiplication** operator is **\***, **/** is the  
980 **division** operator, **%** is the **remainder** operator, and **^** is the **exponent**  
981 operator. MathPiper has more operators in addition to these and some of them  
982 will be covered later.

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

984 In> 5 - 2  
985 Result: 3

986 In> 3\*4  
987 Result: 12

988 In> 30/3  
989 Result: 10

990 In> 11%5  
991 Result: 1

992 In> 2^3  
993 Result: 8

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

995 In> -3  
996 Result: -3

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

999 In> - -3  
1000 Result: 3

1001 In MathPiper, **operators** are symbols (or groups of symbols) that are  
1002 implemented with **procedures**. One can either call the procedure that an  
1003 operator represents directly, or use the operator to call the procedure indirectly.  
1004 However, using operators requires less typing and they often make a program  
1005 easier to read.

### 1006 **9.3 Operator Precedence**

1007 When expressions contain more than one operator, MathPiper uses a set of rules  
1008 called **operator precedence** to determine the order in which the operators are  
1009 applied to the values in the expression. Operator precedence is also referred to  
1010 as the **order of operations**. Operators with higher precedence are evaluated

1011 before operators with lower precedence. The following table shows a subset of  
1012 MathPiper's operator precedence rules with higher precedence operators being  
1013 placed higher in the table:

1014	$\wedge$	Exponents (right associative).
1015	$/$	Then division (left associative).
1016	$*$	Then multiplication (left associative).
1017	$\%$	Then the remainder operator (left associative).
1018	$+, -$	Finally, addition and subtraction (left associative).

1019 This multi-operator expression will be used as an example to illustrate the  
1020 precedence rules.

1021 1) source code form:

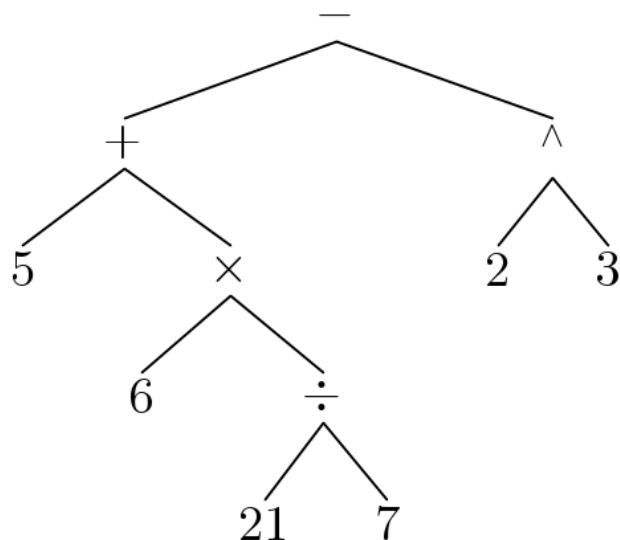
1022 
$$5 + 6 * 21 / 7 - 2^3$$

1023 2) traditional mathematics form:

$$5 + 6 \times \frac{21}{7} - 2^3$$

1024 3) expression tree form:

1025 In> Show(TreeView('(5 + 6\*21/7 - 2^3)))  
1026 Result: class javax.swing.JFrame



1027 The ' operator in the above code is named the “hold” operator, and it prevents an  
 1028 expression from being evaluated. In the following code, the ' operator is used to  
 1029 prevent the expression  $2 + 3$  and the variable 'a' from being evaluated:

1030 In> '(2 + 3)  
 1031 Result: 2+3

1032 In> a := 3  
 1033 Result: 3

1034 In> 'a  
 1035 Result: a

1036 The “hold” operator is useful when one wants to work with an expression instead  
 1037 of the value that the expression returns.

1038 MathPiper uses **post-order** evaluation of expressions instead of **PEMDAS**. This  
 1039 is how post-order evaluation works (See  
 1040 [http://patternmatics.com/test/expression\\_structure.html](http://patternmatics.com/test/expression_structure.html)):

- 1041 1) Start with the operator that is at the top of the expression tree.
- 1042 2) Evaluate the operator's left subtree.
- 1043 3) Evaluate the operator's right subtree.
- 1044 4) Evaluate the operator.

1045 Let's manually apply the precedence rules and post-order evaluation to the multi-  
 1046 operator expression we used earlier.

1047 According to post-order evaluation and the precedence rules, this is the order in  
1048 which MathPiper evaluates the operations in this expression:

```
1049 5 + 6*21/7 - 2^3
1050 5 + 6*3 - 2^3
1051 5 + 18 - 2^3
1052 23 - 2^3
1053 23 - 8
1054 4
```

1055 Starting with the first line, MathPiper evaluates the */* operator first, which  
1056 results in the **3** in the line below it. In the second line, the *\** operator is executed  
1057 next, and so on. The last line shows that the final result after all of the operators  
1058 have been evaluated is **15**.

#### 1059 **9.4 Changing The Order Of Operations In An Expression**

1060 The default order of operations for an expression can be changed by grouping  
1061 various parts of the expression within parentheses **()**. Parentheses force the  
1062 code that is placed inside of them to be evaluated before any other operators are  
1063 evaluated. For example, the expression  $2 + 4*5$  evaluates to 22 using the  
1064 default precedence rules:

```
1065 In> 2 + 4*5
1066 Result: 22
```

1067 If parentheses are placed around  $2 + 4$ , however, the addition operator is forced  
1068 to be evaluated before the multiplication operator and the result is 30:

```
1069 In> (2 + 4)*5
1070 Result: 30
```

1071 Parentheses can also be nested and nested parentheses are evaluated from the  
1072 most deeply nested parentheses outward:

```
1073 In> ((2 + 4)*3)*5
1074 Result: 90
```

1075 (Note: precedence adjusting parentheses are different from the parentheses that  
1076 are used to call procedures.)

1077 Since parentheses are evaluated before any other operators, they are placed at  
1078 the top of the precedence table:

1079     ()     Parentheses are evaluated from the inside out.  
1080     ^     Exponents (right associative).  
1081     /     Then division (left associative).  
1082     \*     Then multiplication (left associative).  
1083     %     Then the remainder operator (left associative).  
1084     +, −   Finally, addition and subtraction (left associative).

## 1085   **9.5 Procedures & Procedure Names**

1086   In programming, **procedures** are named sequences of code that can be executed  
1087   one or more times by being **called** from other parts of the same program or  
1088   called from other programs. Procedures **can have values passed to them** from  
1089   the calling code (called **arguments**), and they **always return a value** back to  
1090   the calling code when they are finished executing. An example of a procedure is  
1091   the **Even?()** procedure, which was discussed in an previous section.

1092   Procedures are one way that MathPiper enables code to be reused. Most  
1093   programming languages allow code to be reused in this way, although in other  
1094   languages these named sequences of code are sometimes called **subroutines**,  
1095   **procedures**, or **methods**.

1096   The procedures that come with MathPiper have names that consist of either a  
1097   single word (such as **Sum()**) or multiple words that have been put together to  
1098   form a compound word (such as **FillList()**). All letters in the names of  
1099   procedures that come with MathPiper are lower case except the beginning letter  
1100   in each word, which are upper case.

## 1101   **9.6 Procedures That Produce Side Effects**

1102   Most procedures are executed to obtain the **results** they produce, but some  
1103   procedures are executed in order to **have them perform work that is not in**  
1104   **the form of a result**. Procedures that perform work that is not in the form of a  
1105   result are said to produce **side effects**. Side effects include many forms of work  
1106   such as sending information to the user, opening files, and changing values in the  
1107   computer's memory.

1108   When a procedure produces a side effect that sends information to the user, this  
1109   information has the words **Side Effects:** placed before it in the output instead of  
1110   the word **Result:**. The **Echo()** and **Write()** procedures are examples of  
1111   procedures that produce side effects, and they are covered in the next section.

## 1112 9.6.1 Printing Related Procedures: Echo(), Write(), And Newline()

1113 The printing related procedures send text information to the user and this is  
1114 usually referred to as "printing" in this document. However, it may also be called  
1115 "echoing" and "writing".

### 1116 9.6.1.1 The Echo() Procedure

1117 The **Echo()** procedure takes one expression (or multiple expressions separated  
1118 by commas) evaluates each expression, and then prints the results as side effect  
1119 output. The following examples illustrate this:

```
1120 In> Echo(1)
1121 Result: True
1122 Side Effects>
1123 1
```

1124 In this example, the number 1 was passed to the Echo() procedure, the number  
1125 was evaluated (all numbers evaluate to themselves), and the result of the  
1126 evaluation was then printed as a side effect. Notice that Echo() **also returned a**  
1127 **result**. In MathPiper, all procedures return a result, but procedures whose main  
1128 purpose is to produce a side effect usually just return a result of **True** if the side  
1129 effect succeeded or **False** if it failed. In this case, Echo() returned a result of  
1130 **True** because it was able to successfully print a 1 as its side effect.

1131 The next example shows multiple expressions being sent to Echo() (notice that  
1132 the expressions are separated by commas):

```
1133 In> Echo(1, 1+2, 2*3)
1134 Result: True
1135 Side Effects>
1136 1 3 6
```

1137 The expressions were each evaluated and their results were returned (separated  
1138 by spaces) as side effect output.

1139 Each time an Echo() procedure is executed, it always forces the display to drop  
1140 down to the next line after it is finished. This can be seen in the following  
1141 program, which is similar to the previous one except it uses a separate Echo()  
1142 procedure to display each expression:

```
1143 %mathpiper
1144 Echo(1);
1145 Echo(1+2);
1146 Echo(2*3);
```

```
1147 %/mathpiper
1148     %output,preserve="false"
1149     Result: True
1150
1151     Side Effects:
1152     1
1153     3
1154     6
1155 .    %/output
```

1156 Notice how the 1, the 3, and the 6 are each on their own line.

1157 Now that we have seen how Echo() works, let's use it to do something useful. If  
1158 more than one expression is evaluated in a %mathpiper fold, only the result from  
1159 the last expression that was evaluated (which is usually the bottommost  
1160 expression) is displayed:

```
1161 %mathpiper
1162 a := 1;
1163 b := 2;
1164 c := 3;
1165 %/mathpiper
1166     %output,preserve="false"
1167     Result: 3
1168 .    %/output
```

1169 In MathPiper, **programs are executed one line at a time, starting at the**  
1170 **topmost line of code and working downwards from there.** In this example,  
1171 the line a := 1; is executed first, then the line b := 2; is executed, and so on.

1172 Notice, however, that even though we wanted to see what was assigned to all  
1173 three variables, only the last variable's value was displayed.

1174 The following example shows how Echo() can be used to display the values that  
1175 are assigned to all three variables:

```
1176 %mathpiper
1177 a := 1;
1178 Echo(a);
1179 b := 2;
1180 Echo(b);
1181 c := 3;
1182 Echo(c);
```

```
1183 %/mathpiper
1184     %output,preserve="false"
1185     Result: True
1186
1187     Side Effects:
1188     1
1189     2
1190     3
1191 . %/output
```

### 1192 9.6.1.2 Echo Procedures Are Useful For "Debugging" Programs

1193 The errors that are in a program are often called "bugs". This name came from  
1194 the days when computers were the size of large rooms and were made using  
1195 electromechanical parts. Periodically, bugs would crawl into the machines and  
1196 interfere with its moving mechanical parts and this would cause the machine to  
1197 malfunction. The bugs needed to be located and removed before the machine  
1198 would run properly again.

1199 Of course, even back then most program errors were produced by programmers  
1200 entering wrong programs or entering programs wrong, but they liked to say that  
1201 all of the errors were caused by bugs and not by themselves! The process of  
1202 fixing errors in a program became known as **debugging** and the names "bugs"  
1203 and "debugging" are still used by programmers today.

1204 One of the standard ways to locate bugs in a program is to place **Echo()**  
1205 procedure calls in the code at strategic places that **print the contents of**  
1206 **variables and display messages**. These Echo() procedures will enable you to  
1207 see what your program is doing while it is running. After you have found and  
1208 fixed the bugs in your program, you can remove the debugging Echo() procedure  
1209 calls or comment them out if you think they may be needed later (comments are  
1210 covered in a later section).

### 1211 9.6.1.3 Write()

1212 The **Write()** procedure is similar to the Echo() procedure except it does not  
1213 automatically drop the display down to the next line after it finishes executing:

```
1214 %mathpiper
1215 Write(1);
1216 Write(2);
1217 Echo(3);
```



```
1218 %/mathpiper
1219     %output,preserve="false"
1220     Result: True
1221
1222     Side Effects:
1223     123
1224 . %/output
```

1225 Write() and Echo() have other differences besides the one discussed here and  
1226 more information about them can be found in the documentation for these  
1227 procedures.

#### 1228 **9.6.1.4 NewLine()**

1229 The **NewLine()** procedure starts a new line in the side effects output. It can be  
1230 used to print blank lines, which are useful for placing vertical space between  
1231 printed lines:

```
1232 %mathpiper
1233 a := 1;
1234 Echo(a);
1235 NewLine();
1236
1237 b := 2;
1238 Echo(b);
1239 NewLine();
1240
1241 c := 3;
1242 Echo(c);
1243
1244 %/mathpiper
```

```
1242     %output,preserve="false"
1243     Result: True
1244
1245     Side Effects:
1246     1
1247
1248     2
1249
1250     3
1251 . %/output
```

### 1250 **9.7 Expressions Are Separated By Semicolons**

1251 As discussed earlier, all of the expressions that are inside of a **%mathpiper** fold

1252 must have a semicolon (;) after them. However, the expressions evaluated in the  
1253 **MathPiper console** did not have a semicolon after them. MathPiper actually  
1254 requires that all expressions end with a semicolon, but one does not need to add  
1255 a semicolon to an expression that is typed into the MathPiper console **because**  
1256 **the console adds it automatically** when the expression is executed.

### 1257 9.7.1 Placing More Than One Expression On A Line In A Fold

1258 All the previous code examples have had each of their expressions on a separate  
1259 line, but multiple expressions can also be placed on a single line because the  
1260 semicolons tell MathPiper where one expression ends and the next one begins:

```
1261 %mathpiper
1262 a := 1; Echo(a); b := 2; Echo(b); c := 3; Echo(c);
1263 %/mathpiper
1264 %output,preserve="false"
1265 Result: True
1266
1267 Side Effects:
1268 1
1269 2
1270 3
1271 . %/output
```

1272 The spaces that are in the code of this example are used to make the code more  
1273 readable. Any spaces that are present within any expressions or between them  
1274 are ignored by MathPiper and if we remove the spaces from the previous code,  
1275 the output remains the same:

```
1276 %mathpiper
1277 a:=1;Echo(a);b:=2;Echo(b);c:=3;Echo(c);
1278 %/mathpiper
1279 %output,preserve="false"
1280 Result: True
1281
1282 Side Effects:
1283 1
1284 2
1285 3
1286 . %/output
```

## 1287 9.7.2 Placing Consecutive Expressions Into A Code Sequence

1288 It is often useful to place a sequence of expressions that are used together to  
1289 accomplish a task into a group. In MathPiper these groups are called "code  
1290 sequences." A **code sequence** (which is also called a **compound expression**)  
1291 consists of one or more expressions that are separated by semicolons and placed  
1292 within an open brace (**{**) and close brace (**}**) pair. When a code sequence is  
1293 evaluated, each expression in the sequence will be executed from left-to-right or  
1294 top-to-bottom. The following example shows expressions being executed within  
1295 a code sequence:

```
1296 In> {a := 1; Echo(a); b := 2; Echo(b); c := 3; Echo(c);}
1297 Result: True
1298 Side Effects>
1299 1
1300 2
1301 3
```

1302 Notice that all of the expressions were executed, and 1-3 was printed as a side  
1303 effect. Code sequences **always return the result of the last expression**  
1304 **executed as the result of the whole sequence**. In this case, **True** was  
1305 returned as the result because the last **Echo(c)** procedure returned **True**. If we  
1306 place **another expression after the Echo(c) procedure**, however, **the**  
1307 **sequence will execute this new expression last and its result will be the**  
1308 **one returned by the sequence**:

```
1309 In> {a := 1; Echo(a); b := 2; Echo(b); c := 3; Echo(c); 2 + 2;}
1310 Result: 4
1311 Side Effects>
1312 1
1313 2
1314 3
```

1315 Finally, code sequences can have their contents placed on separate lines if  
1316 desired:

```
1317 %mathpiper
1318 {
1319     a := 1;
1320     Echo(a);
1321     b := 2;
1322     Echo(b);
1323     c := 3;
```

```
1329     Echo(c);
1330 }

1331 %/mathpiper

1332     %output,preserve="false"
1333     Result: True
1334
1335     Side Effects:
1336     1
1337     2
1338     3
1339 .    %/output
```

1340 Code sequences are very powerful, and we will be discussing them further in  
1341 later sections.

#### 1342 **9.7.2.1 Automatic Bracket, Parentheses, And Brace Match Indicating**

1343 In programming, most open brackets '[' have a close bracket ']', most open  
1344 parentheses '(' have a close parentheses ')', and most open braces '{' have a  
1345 close brace '}'. It is often difficult to make sure that each "open" character has a  
1346 matching "close" character and if any of these characters don't have a match,  
1347 then an error will be produced.

1348 Thankfully, most programming text editors have a character match indicating  
1349 tool that will help locate problems. To try this tool, paste the following code into  
1350 a .mpws file and follow the directions that are present in its comments:

```
1351 %/mathpiper

1352 /*
1353     Copy this code into a .mpws file. Then, place the cursor
1354     to the immediate right of any {, }, [, ], (, or ) character.
1355     You should notice that the match to this character is
1356     indicated by a rectangle being drawing around it.
1357 */

1358 list := [1,2,3];

1359 {
1360     Echo("Hello");

1361     Echo(list);
1362 }

1363 %/mathpiper
```

## 1364 9.8 Strings

1365 A **string** is a **value** that is used to hold text-based information. The typical  
1366 expression that is used to create a string consists of **text that is enclosed**  
1367 **within double quotes**. Text in a program's source code that is enclosed in  
1368 double quotes is called a **string literal**. Strings can be assigned to variables just  
1369 like numbers can, and strings can also be displayed using the Echo() procedure.  
1370 The following program assigns a string value to the variable 'a' and then prints it  
1371 to the user:

```
1372 %mathpiper
1373 a := "Hello, I am a string.";
1374 Echo(a);
1375 %/mathpiper
1376     %output,preserve="false"
1377     Result: True
1378
1379     Side Effects:
1380     Hello, I am a string.
1381 . %/output
```

### 1382 9.8.1 The MathPiper Console and MathPiper Folds Can Access The Same 1383 Variables

1384 A useful aspect of using MathPiper inside of MathPiperIDE is that variables that  
1385 are assigned inside of a **%mathpiper fold** are accessible inside of the  
1386 **MathPiper console** and variables that are assigned inside of the **MathPiper**  
1387 **console** are available inside of **%mathpiper folds**. For example, after the above  
1388 fold is executed, the string that has been assigned to variable 'a' can be  
1389 displayed in the MathPiper console:

```
1390 In> a
1391 Result: "Hello, I am a string."
```

### 1392 9.8.2 Using Strings To Make Echo's Output Easier To Read

1393 When the Echo() procedure is used to print the values of multiple variables, it is  
1394 often helpful to print some information next to each variable so that it is easier to  
1395 determine which value came from which Echo() procedure in the code. The  
1396 following program prints the name of the variable that each value came from  
1397 next to it in the side effects output:

```
1398 %mathpiper
```

```
1399 a := 1;  
1400 Echo("Variable a: ", a);  
  
1401 b := 2;  
1402 Echo("Variable b: ", b);  
  
1403 c := 3;  
1404 Echo("Variable c: ", c);  
  
1405 %/mathpiper  
  
1406     %output,preserve="false"  
1407     Result: True  
1408  
1409     Side Effects:  
1410     Variable a: 1  
1411     Variable b: 2  
1412     Variable c: 3  
1413 . %/output
```

### 1414 9.8.2.1 Combining Strings With The + Operator

1415 If you need to combine two or more strings into one string, you can use the +  
1416 operator like this:

```
1417 In> "A" + "B" + "C"  
1418 Result: "ABC"  
  
1419 In> "Hello " + "there!"  
1420 Result: "Hello there!"
```

### 1421 9.8.2.2 WriteString()

1422 The **WriteString()** procedure prints a string without showing the double quotes  
1423 that are around it. For example, here is the Write() procedure being used to  
1424 print the string "Hello":

```
1425 In> Write("Hello")  
1426 Result: True  
1427 Side Effects:  
1428 "Hello"
```

1429 Notice the double quotes? Here is how the WriteString() procedure prints  
1430 "Hello":

```
1431 In> WriteString("Hello")  
1432 Result: True
```

```
1433 Side Effects:
1434 Hello
```

### 1435 9.8.2.3 *Nl()*

1436 The **Nl()** (New Line) procedure is used with the + procedure to place newline  
1437 characters inside of strings:

```
1438 In> WriteString("A" + Nl() + "B")
1439 Result: True
1440 Side Effects:
1441 A
1442 B
```

### 1443 9.8.2.4 *Space()*

1444 The Space() procedure is used to add spaces to printed output:

```
1445 In> WriteString("A"); Space(5); WriteString("B")
1446 Result: True
1447 Side Effects:
1448 A      B
```

```
1449 In> WriteString("A"); Space(10); WriteString("B")
1450 Result: True
1451 Side Effects:
1452 A          B
```

```
1453 In> WriteString("A"); Space(20); WriteString("B")
1454 Result: True
1455 Side Effects:
1456 A                  B
```

## 1457 9.8.3 Accessing The Individual Letters/Characters In A String

1458 Individual letters in a string (which are also called **characters**) can be accessed  
1459 by placing the character's position number (also called an **index**) inside of  
1460 brackets **[]** after the variable it is assigned to. A character's position is  
1461 determined by its distance from the left side of the string starting at 1. For  
1462 example, in the string "Hello", 'H' is at position 1, 'e' is at position 2, etc. The  
1463 following code shows individual characters in the above string being accessed:

```
1464 In> a := "Hello, I am a string."
1465 Result: "Hello, I am a string."
```

```
1466 In> a[1]
1467 Result: "H"
```

```
1468 In> a[2]
```

1469 Result: "e"

1470 In> a[3]

1471 Result: "l"

1472 In> a[4]

1473 Result: "l"

1474 In> a[5]

1475 Result: "o"

### 1476 **9.8.3.1 Indexing Before The Beginning Of A String Or Past The End Of A String**

1477 Let's see what happens if an index is used that is less than **1** or greater than the  
1478 length of a given string. First, we will assign the string "Hello" to the variable  
1479 'a':

1480 In> a := "Hello"

1481 Result: "Hello"

1482 Then, we'll index the character at position **1** and then the character at position **0**:

1483 In> a[1]

1484 Result: "H"

1485 In> a[0]

1486 Result:

1487 Exception: In procedure "StringMidGet" :

1488 bad argument number 1(counting from 1) :

1489 The offending argument aindex evaluated to 0

1490 In procedure: Nth,

1491 Notice that using an index of **0** resulted in an error.

1492 Next, let's access the character at position **5** (which is the 'o'), and finally the  
1493 character at position **6**:

1494 In> a[5]

1495 Result: "o"

1496 In> a[6]

1497 Result:

1498 Exception: String index out of range: 8

## 1499 **9.9 Comments**

1500 Source code can often be difficult to understand and therefore all programming  
1501 languages provide the ability for **comments** to be included in the code.



1502 Comments are used to explain what the code near them is doing and they are  
1503 usually meant to be read by humans instead of being processed by a computer.  
1504 Therefore, comments are ignored by the computer when a program is executed.  
1505 There are two ways that MathPiper allows comments to be added to source code.  
1506 The first way is by placing two forward slashes // to the left of any text that is  
1507 meant to serve as a comment. The text from the slashes to the end of the line  
1508 the slashes are on will be treated as a comment. Here is a program that contains  
1509 comments that use slashes:

```
1510 %mathpiper
1511 //This is a comment.

1512 x := 2; //The variable x becomes 2.

1513 %/mathpiper

1514     %output,preserve="false"
1515     Result: 2
1516 .    %/output
```

1517 When this program is executed, any text that starts with slashes is ignored.

1518 The second way to add comments to a MathPiper program is by enclosing the  
1519 comments inside of slash-asterisk/asterisk-slash symbols /\* \*/. This option is  
1520 useful when a comment is too large to fit on one line. Any text between these  
1521 symbols is ignored by the computer. This program shows a longer comment that  
1522 has been placed between these symbols:

```
1523 %mathpiper

1524 /*
1525  This is a longer comment and it uses
1526  more than one line. The following
1527  code assigns the number 3 to variable
1528  x and then returns it as a result.
1529 */

1530 x := 3;

1531 %/mathpiper

1532     %output,preserve="false"
1533     Result: 3
1534 .    %/output
```

## 1535 **9.10 How To Tell If MathPiper Has Crashed And What To Do If It Has**

1536 Sometimes code will be evaluated that has one or more unusual errors in it, and  
1537 the errors will cause MathPiper to "crash". Unfortunately, beginners are more  
1538 likely to crash MathPiper than more experienced programmers are because a  
1539 beginner's program is more likely to have errors in it. When MathPiper crashes,  
1540 no harm is done but it will not work correctly after that. **The only way to**  
1541 **recover from a MathPiper crash is to exit MathPiperIDE and then**  
1542 **relaunch it.** All the information in your buffers will be saved and preserved **but**  
1543 **the contents of the console will not be.** Be sure to copy the contents of the  
1544 console into a buffer and then save it before restarting.

1545 One way to tell if MathPiperIDE has crashed is that it will indicate that **there**  
1546 **are errors in lines of code that are actually fine.** If you are receiving an  
1547 error in code that looks okay to you, simply restarting MathPiperIDE may fix the  
1548 problem. If you restart MathPiperIDE and the error is still present, this usually  
1549 means that there really is an error in the code.

## 1550 **9.11 Exercises**

1551 For the following exercises, create a new MathPiperIDE worksheet file called  
1552 **book\_1\_section\_9\_exercises\_<your first name>\_<your last name>.mpws.**  
1553 **(Note: there are no spaces in this file name).** For example, John Smith's  
1554 worksheet would be called:

1555 **book\_1\_section\_9\_exercises\_john\_smith.mpws.**

1556 After this worksheet has been created, place your answer for each exercise that  
1557 requires a fold into its own fold in this worksheet. Place a title attribute in the  
1558 start tag of each fold that indicates the exercise the fold contains the solution to.  
1559 The folds you create should look similar to this one:

```
1560 %mathpiper,title="Exercise 1"
```

```
1561 //Sample fold.
```

```
1562 %/mathpiper
```

1563 If an exercise uses the MathPiper console instead of a fold, copy the work you  
1564 did in the console into a text file so it can be saved.

### 1565 **9.11.1 Exercise 1**

1566 Change the precedence of the following expression using parentheses so that  
1567 it prints 20 instead of 14:

```
1568 2 + 3 * 4
```

1569 **9.11.2 Exercise 2**

1570 Place the following calculations into a single MathPiper fold, and then use  
1571 one Echo() procedure per variable to print the results of the calculations.  
1572 Put strings in the Echo() procedures that indicate which variable each  
1573 calculated value is assigned to:

```
1574 a := (1+2+3+4+5);
1575 b := (1-2-3-4-5);
1576 c := (1*2*3*4*5);
1577 d := (1/2/3/4/5);
```

1578 **9.11.3 Exercise 3**

1579 Place the following calculations into a single MathPiper fold, and then use  
1580 one Echo() procedure to print the results of all the calculations on a  
1581 **single line** (Remember, the Echo() procedure can print multiple values if  
1582 they are separated by **commas**.):

```
1583 a := (2*2*2*2*2);
1584 b := (2^5);
1585 c := (_x^2 * _x^3);
1586 d := (2^2 * 2^3);
```

1587 **9.11.4 Exercise 4**

1588 The following code assigns a string that contains all of the upper case  
1589 letters of the alphabet to the variable **upper**. Each of the three Echo()  
1590 procedures prints an index number and the letter that is at that position  
1591 in the string. Place this code into a fold and then continue the Echo()  
1592 procedures so that all 26 letters and their index numbers are printed

```
1593 upper := "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

1594 Echo(1,upper[1]);
1595 Echo(2,upper[2]);
1596 Echo(3,upper[3]);
```

1597 **9.11.5 Exercise 5**

1598 Use Echo() procedures to print an index number and the character at this  
1599 position for the following string (this is similar to what was done in the  
1600 previous exercise.):

```
1601 extra := "!.@#$%^&*() _+<>,?/{ }[]|-=";";

1602 Echo(1,extra[1]);
1603 Echo(2,extra[2]);
1604 Echo(3,extra[3]);
```

1605 **9.11.6 Exercise 6**

1606 The following program uses strings and index numbers to print a person's  
1607 name. Create a program that uses the three strings from this program to  
1608 print the names of three of your favorite musical bands.

```
1609 %mathpiper
1610 /*
1611     This program uses strings and index numbers to print
1612     a person's name.
1613 */

1614 upper := "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
1615 lower := "abcdefghijklmnopqrstuvwxyz";
1616 extra := ".!@#$%^&*() _+<>,?/{ }[]|\-= ";

1617 //Print "Mary Smith.".
1618 Echo(upper[13],lower[1],lower[18],lower[25],extra[12],upper[19],lower[13],l
1619 ower[9],lower[20],lower[8],extra[1]);

1620 %/mathpiper

1621     %output,preserve="false"
1622     Result: True
1623
1624     Side Effects:
1625     Mary Smith.
1626 .    %/output
```

## 10 Lists

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

One way to create a list is by placing zero or more expressions separated by commas inside of a **pair of brackets []**. When this notation is present in a program's source code, it is called a **list literal**. In the following example, a list is created that contains various expressions and then it is assigned to the variable **exampleList**:

```
In> exampleList := [7,42,"Hello",1/2,_var]
Result: [7,42,"Hello",1/2,_var]
```

```
In> exampleList
Result: [7,42,"Hello",1/2,_var]
```

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

```
In> Length([7,42,"Hello",1/2,_var])
Result: 5
```

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

```
In> exampleList[1]
Result: 7
```

```
In> exampleList[2]
Result: 42
```

```
In> exampleList[3]
Result: "Hello"
```

```
In> exampleList[4]
Result: 1/2
```

```
In> exampleList[5]
Result: _var
```

1661 The **1st** and **2nd** expressions in this list are **integers**, the **3rd** expression is a  
1662 **string**, the **4th** expression is a **rational number** and the **5th** expression is an  
1663 **unassigned variable**.

1664 Lists can also hold other lists as shown in the following example:

```
1665 In> exampleList := [20, 30, [31, 32, 33], 40]
1666 Result: [20,30,[31,32,33],40]
```

```
1667 In> exampleList[1]
1668 Result: 20
```

```
1669 In> exampleList[2]
1670 Result: 30
```

```
1671 In> exampleList[3]
1672 Result: [31,32,33]
```

```
1673 In> exampleList[4]
1674 Result: 40
1675
```

1676 The expression in the **3rd** position in the list is another **list** that contains the  
1677 integers **31**, **32**, and **33**.

1678 An expression in this second list can be accessed by **two sets of brackets**:

```
1679 In> exampleList[3][2]
1680 Result: 32
```

1681 The **3** inside of the first set of brackets accesses the **3rd** member of the **first** list  
1682 and the **2** inside of the second set of brackets accesses the **2nd** member of the  
1683 **second** list.

## 1684 **10.1 Append!()**

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

1685 The **Append!()** procedure adds an expression to the end of a list:

```
1686 In> testList := [21,22,23]
1687 Result: [21,22,23]
```

```
1688 In> Append!(testList, 24)
1689 Result: [21,22,23,24]
```

## 1690 11 Random Integer Values

1691 It is often useful to use random integers in a program. For example, a program  
1692 may need to simulate the rolling of dice in a game. In this section, a procedure  
1693 for randomly obtaining nonnegative integers is discussed along with how to use  
1694 it to simulate the rolling of dice.

### 1695 11.1 Obtaining Random Integers With The RandomInteger() Procedure

1696 One way that a MathPiper program can generate random integers is with the  
1697 **RandomInteger()** procedure. The RandomInteger() procedure takes an integer  
1698 as an argument and it returns a random integer between 1 and the passed in  
1699 integer. The following example shows random integers between 1 and 5  
1700 **inclusive** being obtained from RandomInteger(). **Inclusive** here means that  
1701 both 1 and 5 are included in the range of random integers that may be returned.  
1702 If the word **exclusive** was used instead, this would mean that neither 1 nor 5  
1703 would be in the range.

```
1704 In> RandomInteger(5)
1705 Result: 4
1706 In> RandomInteger(5)
1707 Result: 5
1708 In> RandomInteger(5)
1709 Result: 4
1710 In> RandomInteger(5)
1711 Result: 2
1712 In> RandomInteger(5)
1713 Result: 3
1714 In> RandomInteger(5)
1715 Result: 5
1716 In> RandomInteger(5)
1717 Result: 2
1718 In> RandomInteger(5)
1719 Result: 2
1720 In> RandomInteger(5)
1721 Result: 1
1722 In> RandomInteger(5)
1723 Result: 2
```

1724 Random integers between 1 and 100 can be generated by passing 100 to  
1725 RandomInteger():

```
1726 In> RandomInteger(100)
1727 Result: 15
1728 In> RandomInteger(100)
1729 Result: 14
```

```
1730 In> RandomInteger(100)
1731 Result: 82
1732 In> RandomInteger(100)
1733 Result: 93
1734 In> RandomInteger(100)
1735 Result: 32
```

1736 A range of random integers that does not start with 1 can also be generated by  
1737 using the **two argument** version of **RandomInteger()**. For example, random  
1738 integers between 25 and 75 can be obtained by passing RandomInteger() the  
1739 lowest integer in the range and the highest one:

```
1740 In> RandomInteger(25, 75)
1741 Result: 28
1742 In> RandomInteger(25, 75)
1743 Result: 37
1744 In> RandomInteger(25, 75)
1745 Result: 58
1746 In> RandomInteger(25, 75)
1747 Result: 50
1748 In> RandomInteger(25, 75)
1749 Result: 70
```

## 1750 **11.2 Simulating The Rolling Of Dice**

1751 The following example shows the simulated rolling of a single six sided die using  
1752 the RandomInteger() procedure:

```
1753 In> RandomInteger(6)
1754 Result: 5
1755 In> RandomInteger(6)
1756 Result: 6
1757 In> RandomInteger(6)
1758 Result: 3
1759 In> RandomInteger(6)
1760 Result: 2
1761 In> RandomInteger(6)
1762 Result: 5
```

1763 Code that simulates the rolling of two 6 sided dice can be evaluated in the  
1764 MathPiper console by placing it within a **code sequence**. The following code  
1765 outputs the sum of the two simulated dice:

```
1766 In> {a := RandomInteger(6); b := RandomInteger(6); a + b;}
1767 Result: 6
1768 In> {a := RandomInteger(6); b := RandomInteger(6); a + b;}
1769 Result: 12
1770 In> {a := RandomInteger(6); b := RandomInteger(6); a + b;}
1771 Result: 6
```



```
1772 In> {a := RandomInteger(6); b := RandomInteger(6); a + b;}
1773 Result: 4
1774 In> {a := RandomInteger(6); b := RandomInteger(6); a + b;}
1775 Result: 3
1776 In> {a := RandomInteger(6); b := RandomInteger(6); a + b;}
1777 Result: 8
```

1778 Now that we have the ability to simulate the rolling of two 6 sided dice, it would  
1779 be interesting to determine if some sums of these dice occur more frequently  
1780 than other sums. What we would like to do is to roll these simulated dice  
1781 hundreds (or even thousands) of times and then analyze the sums that were  
1782 produced. We don't have the programming capability to easily do this yet, but  
1783 after we finish the section on **While loops**, we will.

### 1784 **11.3 Exercises**

1785 For the following exercises, create a new MathPiperIDE worksheet file called  
1786 **book\_1\_section\_11\_exercises\_<your first name>\_<your last name>.mpws**  
1787 (**Note: there are no spaces in this file name**). For example, John Smith's  
1788 worksheet would be called:

1789 **book\_1\_section\_11\_exercises\_john\_smith.mpws.**

1790 After this worksheet has been created, place your answer for each exercise that  
1791 requires a fold into its own fold in this worksheet. Place a title attribute in the  
1792 start tag of each fold that indicates the exercise the fold contains the solution to.  
1793 The folds you create should look similar to this one:

```
1794 %mathpiper,title="Exercise 1"
```

```
1795 //Sample fold.
```

```
1796 %/mathpiper
```

1797 If an exercise uses the MathPiper console instead of a fold, copy the work you  
1798 did in the console into a text file so it can be saved.

#### 1799 **11.3.1 Exercise 1**

1800 Create a program that will roll two simulated dice 20 times, and print the  
1801 results of each of these rolls.

## 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 that have the ability to make decisions are able to solve a wider range of problems than programs that can't make decisions.

### 12.1 Relational Operators

A program's decision making ability is based on a set of special operators that are called **relational operators**. Another name for them is **comparison operators**, but we will call them relational operators in this book. A **relational operator** is an operator that is used to **compare two values**. Expressions that contain relational operators return a **boolean value** and a **boolean value** is one that can only be **True** or **False**. When the words "True" and "False" are present in a program's source code, they are called **boolean literals**. In case you are curious about the strange name, boolean values come from the area of mathematics called **boolean logic**. This logic was created by a mathematician named **George Boole** and this is where the name boolean came from. Table 2 shows the relational operators that MathPiper uses:

Operator	Description
<code>x ==? y</code>	Returns <b>True</b> if the two values are equal and <b>False</b> if they are not equal. Notice that <code>==?</code> performs a comparison and not an assignment like <code>:=</code> does.
<code>x !=? y</code>	Returns <b>True</b> if the values are not equal and <b>False</b> if they are equal.
<code>x &lt;? y</code>	Returns <b>True</b> if the left value is less than the right value and <b>False</b> if the left value is not less than the right value.
<code>x &lt;=? y</code>	Returns <b>True</b> if the left value is less than or equal to the right value and <b>False</b> if the left value is not less than or equal to the right value.
<code>x &gt;? y</code>	Returns <b>True</b> if the left value is greater than the right value and <b>False</b> if the left value is not greater than the right value.
<code>x &gt;=? y</code>	Returns <b>True</b> if the left value is greater than or equal to the right value and <b>False</b> if the left value is not greater than or equal to the right value.

Table 2: Relational Operators

This example shows some of these relational operators being evaluated in the MathPiper console:

```
1823 In> 1 <? 2
1824 Result: True
```

```
1825 In> 4 >? 5
1826 Result: False
```

```
1827 In> 8 >=? 8
1828 Result: True
```

```
1829 In> 5 <=? 10
1830 Result: True
```

1831 The following examples show each of the relational operators in Table 2 being  
 1832 used to compare values that have been assigned to variables **x** and **y**:

```
1833 %mathpiper
```

```
1834 // Example 1.
```

```
1835 x := 2;
1836 y := 3;
```

```
1837 Echo(x, "=? ", y, ": ", x ==? y);
1838 Echo(x, "!=? ", y, ": ", x !=? y);
1839 Echo(x, "<? ", y, ": ", x <? y);
1840 Echo(x, "<=? ", y, ": ", x <=? y);
1841 Echo(x, ">? ", y, ": ", x >? y);
1842 Echo(x, ">=? ", y, ": ", x >=? y);
```

```
1843 %/mathpiper
```

```
1844 %output,preserve="false"
1845 Result: True
```

```
1846
1847 Side Effects:
1848 2 ==? 3 : False
1849 2 !=? 3 : True
1850 2 <? 3 : True
1851 2 <=? 3 : True
1852 2 >? 3 : False
1853 2 >=? 3 : False
```

```
1854 . %/output
```

```
1855 %mathpiper
```

```
1856 // Example 2.
```

```
1857 x := 2;
1858 y := 2;
```

```
1859 Echo(x, "=? ", y, ": ", x ==? y);
```

```

1860     Echo(x, "!=?" , y, ":" , x !=? y);
1861     Echo(x, "<?" , y, ":" , x <? y);
1862     Echo(x, "<=?" , y, ":" , x <=? y);
1863     Echo(x, ">?" , y, ":" , x >? y);
1864     Echo(x, ">=?" , y, ":" , x >=? y);

```

```
1865 %/mathpiper
```

```

1866     %output,preserve="false"
1867     Result: True
1868
1869     Side Effects:
1870     2 =? 2 : True
1871     2 !=? 2 : False
1872     2 <? 2 : False
1873     2 <=? 2 : True
1874     2 >? 2 : False
1875     2 >=? 2 : True
1876 . %/output

```

```
1877 %mathpiper
```

```

1878 // Example 3.
1879 x := 3;
1880 y := 2;

```

```

1881 Echo(x, "=?" , y, ":" , x =? y);
1882 Echo(x, "!=?" , y, ":" , x !=? y);
1883 Echo(x, "<?" , y, ":" , x <? y);
1884 Echo(x, "<=?" , y, ":" , x <=? y);
1885 Echo(x, ">?" , y, ":" , x >? y);
1886 Echo(x, ">=?" , y, ":" , x >=? y);

```

```
1887 %/mathpiper
```

```

1888     %output,preserve="false"
1889     Result: True
1890
1891     Side Effects:
1892     3 =? 2 : False
1893     3 !=? 2 : True
1894     3 <? 2 : False
1895     3 <=? 2 : False
1896     3 >? 2 : True
1897     3 >=? 2 : True
1898 . %/output

```

1899 Relational operators are placed at a lower level of precedence than the other  
 1900 operators we have covered to this point:

1901     ()     Parentheses are evaluated from the inside out.  
1902     ^     Exponents (right associative).  
1903     /     Then division (left associative).  
1904     \*     Then multiplication (left associative).  
1905     %     Then the remainder operator (left associative).  
1906     +, -   Addition and subtraction (left associative).  
1907     =?,!=?,<?,<=?,>?,>=? Finally, relational operators are evaluated (left  
1908            associative).

## 1909 **12.2 Predicate Expressions**

1910 Expressions that return either **True** or **False** are called "**predicate**" expressions.  
1911 By themselves, predicate expressions are not very useful. They only become so  
1912 when they are used with special decision making procedures, like the If()  
1913 procedure (which is discussed in the next section).

## 1914 **12.3 Exercises**

1915 For the following exercises, create a new MathPiperIDE worksheet file called  
1916 **book\_1\_section\_12a\_exercises\_<your first name>\_<your last**  
1917 **name>.mpws**. (**Note: there are no spaces in this file name**). For example,  
1918 John Smith's worksheet would be called:

1919 **book\_1\_section\_12a\_exercises\_john\_smith.mpws**.

1920 After this worksheet has been created, place your answer for each exercise that  
1921 requires a fold into its own fold in this worksheet. Place a title attribute in the  
1922 start tag of each fold that indicates the exercise the fold contains the solution to.  
1923 The folds you create should look similar to this one:

1924 `%mathpiper,title="Exercise 1"`

1925 `//Sample fold.`

1926 `%/mathpiper`

1927 If an exercise uses the MathPiper console instead of a fold, copy the work you  
1928 did in the console into a text file so it can be saved.

### 1929 **12.3.1 Exercise 1**

1930 Open a MathPiper session and evaluate the following predicate expressions:

1931 In> 3 =? 3

1932 In> 3 =? 4

1933 In> 3 <? 4

1934 In> 3 !=? 4

1935 In> -3 <? 4

1936 In> 4 >=? 4

1937 In> 1/2 <? 1/4

1938 In> 15/23 <? 122/189

1939 /\*In the following two expressions, notice that 1/2 is not considered to be  
1940 equal to .5 unless it is converted to a numerical value first.\*/

1941 In> 1/2 =? .5

1942 In> NM(1/2) =? .5

### 1943 12.3.2 Exercise 2

1944 Come up with 10 predicate expressions of your own and evaluate them in the  
1945 MathPiper console.

## 1946 12.4 Making Decisions With The If() Procedure & Predicate Expressions

1947 Most programming languages have the ability to make decisions, and the most  
1948 commonly used procedure for making decisions in MathPiper is the **If()**  
1949 procedure:

1950 With code sequence body:

1951 If(predicate)

1952 {

1953     body\_expressions

1954 }

Notice that in bodied procedures, the ; is placed after the closing }, not after the closing ).

1955 Without code sequence body:

1956 If(predicate) body\_expression;

1957 The expression or expressions that are contained in a If() procedure are called its  
1958 "**body**", and all procedures that have bodies are called "**bodied**" procedures. If a  
1959 body contains more than one expression, then these expressions need to be

placed within a **code sequence** (code sequences were discussed in an earlier section). What a procedure's body is will become clearer after studying some example programs.

The way the If() procedure works is it evaluates the "**predicate**" expression that is passed to it as an argument, and then it looks at the value that the expression returns. If this value is **True**, the body of the If() procedure is executed. If the predicate expression evaluates to **False**, the body is not executed. (Note: any procedure that accepts a predicate expression as a parameter can also accept the boolean values True and False).

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

```

1972 %mathpiper
1973 number := 6;
1974 If(number >? 5)
1975 {
1976     Echo(number, "is greater than 5.");
1977 }
1978 Echo("End of program.");
1979 %/mathpiper
1980     %output,preserve="false"
1981     Result: True
1982
1983     Side Effects:
1984     6 is greater than 5.
1985     End of program.
1986 . %/output

```

In this program, number has been set to 6 and therefore the expression number >? 5 is **True**. When the **If()** procedure evaluates the **predicate expression** and determines it is **True**, it then executes the **Echo()** procedure that is **in its body**. The **second Echo()** procedure at the bottom of the program prints "End of program" regardless of what the If() procedure does.

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

```

1993 %mathpiper
1994 number := 4;
1995 If(number >? 5)
1996 {

```

```
1997     Echo(number, "is greater than 5.");
1998 }
```

```
1999 Echo("End of program.");
```

```
2000 %/mathpiper
```

```
2001     %output,preserve="false"
```

```
2002     Result: True
```

```
2003
```

```
2004     Side Effects:
```

```
2005     End of program.
```

```
2006 .    %/output
```

2007 This time the expression **number >? 5** returns a value of **False**, which causes  
2008 the **If()** procedure to not execute its body.

2009 This version of the program contains an If() procedure that does not use a code  
2010 sequence as a body:

```
2011 %mathpiper
```

```
2012 number := 4;
```

```
2013 If(number >? 5) Echo(number, "is greater than 5.");
```

```
2014 Echo("End of program.");
```

```
2015 %/mathpiper
```

```
2016     %output,preserve="false"
```

```
2017     Result: True
```

```
2018
```

```
2019     Side Effects:
```

```
2020     End of program.
```

```
2021 .    %/output
```

2022 If the **:=** operator is used in the body of an If() procedure **that does not use a**  
2023 **code sequence for its body**, the unbodied expression must be placed within  
2024 parentheses:

```
2025 %mathpiper
```

```
2026 number := 6;
```

```
2027 If(number >? 5) (number := 0);
```



```
2028 number;  
2029 %/mathpiper  
2030     %output,preserve="false"  
2031     Result: 0  
2032 .    %/output
```

### 2033 12.4.1 One If() Procedure Used With One Else Operator

2034 An If() procedure can be used with an **Else** operator to evaluate one body if a  
2035 predicate expression is True, and an alternative body if the predicate expression  
2036 is False. The format for If/Else code is as follows:

```
2037 If(predicate)  
2038 {  
2039     evaluate_this_body_if_True.  
2040 }  
2041 Else  
2042 {  
2043     evaluate_this_body_if_False  
2044 }
```

2045 The following program prints "4 is NOT greater than 5" because the predicate x  
2046 >? 5 is False:

```
2047 %mathpiper  
2048 x := 4;  
2049 If(x >? 5)  
2050 {  
2051     Echo(x, "is greater than 5.");  
2052 }  
2053 Else  
2054 {  
2055     Echo(x, "is NOT greater than 5.");  
2056 }  
2057 Echo("End of program.");  
2058 %/mathpiper  
2059     %output,preserve="false"  
2060     Result: True  
2061  
2062     Side Effects:  
2063     4 is NOT greater than 5.
```

```
2064     End of program.
2065 .    %/output
```

## 2066 **12.5 The &?, |?, And !? Boolean Operators**

### 2067 **12.5.1 The &? "And" Operator**

2068 Sometimes a programmer needs to check if two expressions are **True** and one  
 2069 way to do this is with the **&?** operator (which is read "**and**"). This is the calling  
 2070 format for the &? operator:

```
expression1 &? expression2
```

2071 If **both** of these expressions return a value of **True**, the **&?** operator will also  
 2072 return a **True**. However, if **either** of the expressions return a **False**, then the &  
 2073 operator will return a **False**. This can be seen in the following example:

```
2074 In> True &? True
2075 Result: True
```

```
2076 In> True &? False
2077 Result: False
```

```
2078 In> False &? True
2079 Result: False
```

```
2080 In> False &? False
2081 Result: False
```

```
2082 In> True &? True &? True &? True
2083 Result: True
```

2084 The following program demonstrates the &? operator being used:

```
2085 %mathpiper
```

```
2086 a := 7;
2087 b := 9;
```

```
2088 Echo("1: ", a <? 5 &? b <? 10);
2089 Echo("2: ", a >? 5 &? b >? 10);
2090 Echo("3: ", a <? 5 &? b >? 10);
2091 Echo("4: ", a >? 5 &? b <? 10);
```

```
2092 If(a >? 5 &? b <? 10)
2093 {
```

```
2094     Echo("These expressions are both true.");
2095 }
2096 %/mathpiper
2097     %output,preserve="false"
2098     Result: True
2099
2100     Side Effects:
2101     1: False
2102     2: False
2103     3: False
2104     4: True
2105     These expressions are both true.
2106 .    %/output
```

## 2107 12.5.2 The |? "Or" Operator

2108 The |? operator (which is read "**or**") is similar to the &? operator in that it only  
2109 works with predicate expressions. However, instead of requiring that both  
2110 expressions be **True** in order to return a **True**, |? will return a **True** if **one or**  
2111 **both expressions are True**.

2112 Here is the calling format for |?:

```
expression1 |? expression2
```

2113 This example shows the |? operator being used:

```
2114 In> True |? True
2115 Result: True
2116 In> True |? False
2117 Result: True
2118 In> False |? True
2119 Result: True
2120 In> False |? False
2121 Result: False
2122 In> False |? False |? True |? False
2123 Result: True
```

2124 The following program also demonstrates the |? operator being used:

```
2125 %mathpiper
```

```

2126 a := 7;
2127 b := 9;

2128 Echo("1: ", a <? 5 |? b <? 10);
2129 Echo("2: ", a >? 5 |? b >? 10);
2130 Echo("3: ", a >? 5 |? b <? 10);
2131 Echo("4: ", a <? 5 |? b >? 10);

2132 If(a <? 5 |? b <? 10)
2133 {
2134     Echo("At least one of these expressions is true.");
2135 }

2136 %/mathpiper

2137     %output,preserve="false"
2138     Result: True
2139
2140     Side Effects:
2141     1: True
2142     2: True
2143     3: True
2144     4: False
2145     At least one of these expressions is true.
2146 . %/output

```

### 2147 12.5.3 The !? "Not" Operator

2148 The !? operator (which is read "not") works with predicate expressions like the  
 2149 &? and |? operators do, except it can only accept **one** expression as input. The  
 2150 way !? works is that it changes a **True** value to a **False** value and a **False** value  
 2151 to a **True** value. Here is the !? operator's calling format:

```
!? expression
```

2152 These are examples of Not> being used:

```

2153 In> !? True
2154 Result: False

2155 In> !? False
2156 Result: True

```

2157 The following is a program that uses the !? operator:

```
2158 %/mathpiper
```

```

2159 Echo("3 =? 3 is ", 3 =? 3);
2160 Echo("!? 3 =? 3 is ", !? 3 =? 3);
2161 %/mathpiper
2162     %output,preserve="false"
2163     Result: True
2164
2165     Side Effects:
2166     3 =? 3 is True
2167     !? 3 =? 3 is False
2168 .    %/output

```

## 2169 12.6 Exercises

2170 For the following exercises, create a new MathPiperIDE worksheet file called  
 2171 **book\_1\_section\_12c\_exercises\_<your first name>\_<your last**  
 2172 **name>.mpws**. (**Note: there are no spaces in this file name**). For example,  
 2173 John Smith's worksheet would be called:

2174 **book\_1\_section\_12c\_exercises\_john\_smith.mpws**.

2175 After this worksheet has been created, place your answer for each exercise that  
 2176 requires a fold into its own fold in this worksheet. Place a title attribute in the  
 2177 start tag of each fold that indicates the exercise the fold contains the solution to.  
 2178 The folds you create should look similar to this one:

```

2179 %mathpiper,title="Exercise 1"
2180 //Sample fold.
2181 %/mathpiper

```

2182 If an exercise uses the MathPiper console instead of a fold, copy the work you  
 2183 did in the console into a text file so it can be saved.

### 2184 12.6.1 Exercise 1

2185 Write a program that uses the RandomInteger() procedure to simulate the  
 2186 flipping of a coin (Hint: you can use 1 to represent a head and 2 to  
 2187 represent a tail.) Use predicate expressions, the If() procedure, and the  
 2188 Echo() procedure to print the string "The coin came up heads." or the  
 2189 string "The coin came up tails.", depending on what the simulated coin flip  
 2190 came up as when the code was executed.

2191 **12.6.2 Exercise 2**

2192 The following program simulates the rolling of two dice and prints a  
2193 message if **both** of the two dice come up less than or equal to 3. Create a  
2194 similar program that simulates the flipping of two coins and print the  
2195 message "Both coins came up heads." if both coins come up heads.

```
2196 %mathpiper
2197 /*
2198     This program simulates the rolling of two dice and prints a message if
2199     both of the two dice come up less than or equal to 3.
2200 */
```

```
2201 die1 := RandomInteger(6);
2202 die2 := RandomInteger(6);

2203 Echo("Die1: ", die1, " Die2: ", die2);
2204 NewLine();

2205 If( die1 <=? 3 &? die2 <=? 3)
2206 {
2207     Echo("Both dice came up <=? to 3.");
2208 }
```

```
2209 %/mathpiper
```

2210 **12.6.3 Exercise 3**

2211 The following program simulates the rolling of two dice and prints a  
2212 message if **either** of the two dice come up less than or equal to 3. Create  
2213 a **similar** program that simulates the flipping of two coins and print the  
2214 message "At least one coin came up heads." if at least one coin comes up  
2215 heads.

```
2216 %mathpiper
2217 /*
2218     This program simulates the rolling of two dice and prints a message if
2219     either of the two dice come up less than or equal to 3.
2220 */
```

```
2221 die1 := RandomInteger(6);
2222 die2 := RandomInteger(6);

2223 Echo("Die1: ", die1, " Die2: ", die2);
2224 NewLine();

2225 If( die1 <=? 3 |? die2 <=? 3)
2226 {
2227     Echo("At least one die came up <=? 3.");
2228 }
```

```
2229 %/mathpiper
```

## 2230 13 The While() And Until() Looping Procedures

### 2231 13.1 The While() Looping Procedure

2232 Many kinds of machines, including computers, derive much of their power from  
 2233 the principle of **repeated cycling**. **Repeated cycling** in a MathPiper program  
 2234 means to execute one or more expressions over and over again and this process  
 2235 is called "**looping**". MathPiper provides a number of ways to implement **loops**  
 2236 in a program and these ways range from straight-forward to subtle.

2237 We will begin discussing looping in MathPiper by starting with the straight-  
 2238 forward **While** procedure. The calling format for the **While** procedure is as  
 2239 follows:

```
2240 While(predicate)
2241 {
2242     body_expressions
2243 }
```

Notice that in bodied procedures, the ; is placed after the closing }, not after the closing ).

2244 The **While** procedure is similar to the **If()** procedure, except it will repeatedly  
 2245 execute the expressions in its body as long as its "predicate" expression is **True**.  
 2246 As soon as the predicate expression returns a **False**, the While() procedure skips  
 2247 the expressions it contains and execution continues with the expression that  
 2248 immediately follows the While() procedure (if there is one).

#### 2249 13.1.1 Printing The Integers From 1 to 10

2250 The following program uses a While() procedure to print the integers from 1 to  
 2251 10:

```
2252 %mathpiper

2253 // This program prints the integers from 1 to 10.

2254 /*
2255     Initialize the variable count to 1
2256     outside of the While "loop".
2257 */
2258 count := 1;

2259 While(count <=? 10)
2260 {
2261     Echo(count);
2262
2263     count := (count + 1); //Increment count by 1.
```

```
2264 }
2265 %/mathpiper
2266     %output,preserve="false"
2267     Result: True
2268
2269     Side Effects:
2270     1
2271     2
2272     3
2273     4
2274     5
2275     6
2276     7
2277     8
2278     9
2279     10
2280 . %/output
```

2281 In this program, a single variable called **count** is created. It is used to tell the  
2282 Echo() procedure which integer to print, and it is also used in the predicate  
2283 expression that determines if the While() procedure should continue to **loop** or  
2284 not.

2285 When the program is executed, **1** is assigned to **count**, and then the While()  
2286 procedure is called. Notice that **1** is assigned to the variable **count** above the  
2287 **While loop**. Assigning an initial value to a variable is called **initializing** the  
2288 variable and in this case, count needs to be initialized before it is used in the  
2289 While() procedure. The predicate expression **count** <=? **10** becomes **1** <=? **10**  
2290 and, since 1 is indeed less than or equal to 10, a value of **True** is returned by the  
2291 predicate expression.

2292 The While() procedure sees that the predicate expression returned a **True** and  
2293 therefore it executes all of the expressions inside of its **body** from top to bottom.

2294 The Echo() procedure prints the current contents of count (which is 1) and then  
2295 the expression count := (count + 1) is executed.

2296 The expression **count := (count + 1)** is a standard expression form that is used  
2297 in many programming languages. Each time an expression in this form is  
2298 evaluated, it **increases the variable it contains by 1**. Another way to describe  
2299 the effect this expression has on **count** is to say that it **increments count by 1**.

2300 In this case **count** contains **1** and, after the expression is evaluated, **count**  
2301 contains **2**.

2302 After the last expression inside the body of the While() procedure is executed,  
2303 the While() procedure reevaluates its predicate expression to determine whether  
2304 it should continue looping or not. Since **count** is **2** at this point, the predicate  
2305 expression returns **True** and the code inside the body of the While() procedure is



2306 executed again. This loop will be repeated until **count** is incremented to **11** and  
2307 the predicate expression returns **False**.

### 2308 13.1.2 Placing The Integers From 1 to 50 In A List

2309 The previous program can be adjusted in a number of ways to achieve different  
2310 results. For example, the following program places the integers from 1 to 50 into  
2311 a list by changing the **10** in the predicate expression to **50** and changing the  
2312 Write procedure to a **Append!()** procedure.

```
2313 %mathpiper
2314 // Place the integers from 1-50 in a list.
2315 integersList := [];
2316 count := 1;
2317 while(count <=? 50)
2318 {
2319     Append!(integersList, count);
2320
2321     count := (count + 1); //Increment count by 1.
2322 }
2323 integersList;
2324 %/mathpiper
2325     %output
2326     Result:
2327 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28
2328 , 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50]
2329 . %/output
```

2330 (Note: In MathPiperIDE, the above numbers will all be on a single line.)

### 2331 13.1.3 Printing The Odd Integers From 1 To 99

2332 The following program prints the odd integers from 1 to 99 by changing the  
2333 **increment value** in the increment expression from **1** to **2**:

```
2334 %mathpiper
2335 //Print the odd integers from 1 to 99.
2336 x := 1;
```

```

2337 While(x <=? 100)
2338 {
2339     Write(x, ', ');
2340     x := (x + 2); //Increment x by 2.
2341 }
2342 %/mathpiper
2343     %output,preserve="false"
2344     Result: True
2345
2346     Side Effects:
2347     1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43,
2348     45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71, 73, 75, 77, 79, 81, 83,
2349     85, 87, 89, 91, 93, 95, 97, 99
2350 .    %/output

```

#### 2351 13.1.4 Placing The Integers From 1 To 100 In Reverse Order Into A List

2352 Finally, the following program prints the integers from 1 to 100 in reverse order:

```

2353 %mathpiper
2354 // Place the integers from 1 to 100 in reverse order into a list.
2355 resultList := [];
2356 x := 100;
2357 While(x >=? 1)
2358 {
2359     Append!(resultList, x);
2360     x := (x - 1); //Decrement x by 1.
2361 }
2362 resultList;
2363 %/mathpiper
2364     %output
2365     Result:
2366     [100, 99, 98, 97, 96, 95, 94, 93, 92, 91, 90, 89, 88, 87, 86, 85, 84, 83, 82, 81, 80, 79, 78, 77, 7
2367     6, 75, 74, 73, 72, 71, 70, 69, 68, 67, 66, 65, 64, 63, 62, 61, 60, 59, 58, 57, 56, 55, 54, 53, 52, 5
2368     1, 50, 49, 48, 47, 46, 45, 44, 43, 42, 41, 40, 39, 38, 37, 36, 35, 34, 33, 32, 31, 30, 29, 28, 27, 2
2369     6, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
2370 .    %/output

```

2371 In order to achieve the reverse ordering, this program had to initialize (which  
2372 means to assign an initial value to a variable) **x** to **100**, check to see if **x** was  
2373 **greater than or equal to 1** ( $x \geq 1$ ), and **decrement** **x** by **subtracting 1**  
2374 **from it** instead of adding 1 to it.

## 2375 13.2 The Until() Looping Procedure

2376 The While() procedure evaluates the predicate expression that is passed to it,  
2377 and then it evaluates its body if the predicate is **True**, and it does not evaluate its  
2378 body if its predicate is **False**. The Until() procedure is similar to the While()  
2379 procedure, except it evaluates its body before it evaluates the predicate  
2380 expression that is passed to it, and it continues looping until the predicate  
2381 expression becomes **True** instead of **False**. Since Until() evaluates its body  
2382 before it evaluates the predicate expression, its body is always evaluated at least  
2383 once.

2384 The calling format for the **Until** procedure is as follows:

```
2385 Until(predicate)
2386 {
2387     body_expressions
2388 }
```

Notice that in bodied procedures, the ; is placed after the closing }, not after the closing ).

### 2389 13.2.1 Printing The Integers From 1 to 10

2390 The following program uses a Until() procedure to print the integers from 1 to  
2391 10:

```
2392 %mathpiper
2393 // This program prints the integers from 1 to 10.
2394 /*
2395     Initialize the variable count to 1
2396     outside of the Until "loop".
2397 */
2398 count := 1;
2399 Until(count == 11)
2400 {
2401     Echo(count);
2402     count := (count + 1); //Increment count by 1.
2403 }
2404
2405 %/mathpiper
```

```
2406 %output,preserve="false"
2407 Result: True
2408
2409 Side Effects:
2410 1
2411 2
2412 3
2413 4
2414 5
2415 6
2416 7
2417 8
2418 9
2419 10
2420 . %/output
```

### 2421 **13.3 Expressions Inside Of Code Sequences Are Indented**

2422 In the programs in the previous sections that use While loops, notice that the  
2423 expressions that are inside of the While() procedure's code sequence are  
2424 **indented**. These expressions do not need to be indented to execute properly,  
2425 but doing so makes the program easier to read.

### 2426 **13.4 Long-Running Loops, Infinite Loops, & Interrupting Execution**

2427 It is easy to create a loop that will execute a **large number of times**, or even **an**  
2428 **infinite number of times**, either on purpose or by mistake. When you execute  
2429 a program that contains an **infinite loop**, it will run until you tell MathPiper to  
2430 **interrupt** its execution. This is done by opening the MathPiper **console** and  
2431 then pressing the "**Halt Evaluation**" button, which in the upper left corner of  
2432 the console.

2433 Let's experiment with the **Halt Evaluation** button by executing a program that  
2434 contains an infinite loop and then stopping it:

```
2435 %mathpiper
2436 //Infinite loop example program.
2437 x := 1;
2438 while(x <= 10)
2439 {
2440     x := 3; //Oops, x is not being incremented!.
2441 }
2442 %/mathpiper
```

```
2443 %output,preserve="false"
2444 Processing...
2445 . %/output
```

2446 Since the contents of `x` is never changed inside the loop, the expression `x <? 10`  
2447 always evaluates to **True**, which causes the loop to continue looping. Notice that  
2448 the %output fold contains the word "**Processing...**" to indicate that the program  
2449 is still running the code.

2450 Execute this program now and then interrupt it using the **Halt Evaluation**  
2451 button. When the program is interrupted, the %output fold will display the  
2452 message "**User halted evaluation**" to indicate that the program was  
2453 interrupted. After a program has been interrupted, the program can be edited  
2454 and then rerun.

### 2455 **13.5 A Program That Simulates Rolling Two Dice 50 Times**

2456 The following program is larger than the previous programs that have been  
2457 discussed in this book, but it is also more interesting and more useful. It uses a  
2458 While() loop to simulate the rolling of two dice 50 times, and it records how  
2459 many times each possible sum has been rolled so that this data can be printed.  
2460 The comments in the code explain what each part of the program does.  
2461 (Remember, if you copy this program to a MathPiperIDE worksheet, you can use  
2462 **rectangular selection mode** to easily remove the line numbers).

```
2463 %mathpiper
2464 /*
2465  This program simulates rolling two dice 50 times.
2466 */

2467 /*
2468  These variables are used to record how many times
2469  a possible sum of two dice has been rolled. They are
2470  all initialized to 0 before the simulation begins.
2471 */
2472 numberOfTwosRolled := 0;
2473 numberOfThreesRolled := 0;
2474 numberOfFoursRolled := 0;
2475 numberOfFivesRolled := 0;
2476 numberOfSixesRolled := 0;
2477 numberOfSevensRolled := 0;
2478 numberOfEightsRolled := 0;
2479 numberOfNinesRolled := 0;
2480 numberOfTensRolled := 0;
2481 numberOfElevenRolled := 0;
2482 numberOfTwelvesRolled := 0;
```

```
2483 Echo("These are the rolls:");

2484 //This variable keeps track of the number of the current roll.
2485 roll := 1;

2486 /*
2487  The simulation is performed inside of this While loop. The number of
2488  times the dice will be rolled can be changed by changing the number 50,
2489  which is in the While procedure's predicate expression.
2490 */
2491 While(roll <=? 50)
2492 {
2493     //Roll the dice.
2494     die1 := RandomInteger(6);
2495     die2 := RandomInteger(6);
2496
2497
2498     //Calculate the sum of the two dice.
2499     rollSum := (die1 + die2);
2500
2501
2502     /*
2503      Print the sum that was rolled. Note: if a large number of rolls
2504      is going to be performed (say >? 1000), it would be best to comment
2505      out this Write() procedure so that it does not put too much text
2506      into the output fold.
2507     */
2508     Write(rollSum, ',');
2509
2510
2511     /*
2512      These If() procedures determine which sum was rolled and then add
2513      1 to the variable that is keeping track of the number of times
2514      that sum was rolled. The bodies of these If() procedures are not in
2515      code sequences.
2516     */
2517     If(rollSum ==? 2) (numberOfTwosRolled := (numberOfTwosRolled + 1));
2518     If(rollSum ==? 3) (numberOfThreesRolled := (numberOfThreesRolled + 1));
2519     If(rollSum ==? 4) (numberOfFoursRolled := (numberOfFoursRolled + 1));
2520     If(rollSum ==? 5) (numberOfFivesRolled := (numberOfFivesRolled + 1));
2521     If(rollSum ==? 6) (numberOfSixesRolled := (numberOfSixesRolled + 1));
2522     If(rollSum ==? 7) (numberOfSevensRolled := (numberOfSevensRolled + 1));
2523     If(rollSum ==? 8) (numberOfEightsRolled := (numberOfEightsRolled + 1));
2524     If(rollSum ==? 9) (numberOfNinesRolled := (numberOfNinesRolled + 1));
2525     If(rollSum ==? 10) (numberOfTensRolled := (numberOfTensRolled + 1));
2526     If(rollSum ==? 11) (numberOfElevensRolled := (numberOfElevensRolled+1));
2527     If(rollSum ==? 12) (numberOfTwelvesRolled := (numberOfTwelvesRolled+1));
2528
```

```

2529
2530     //Increment the roll variable to the next roll number.
2531     roll := (roll + 1);
2532 }

2533 //Print the contents of the sum count variables for visual analysis.
2534 NewLine();
2535 NewLine();
2536 Echo("Number of Twos rolled: ", numberOfTwosRolled);
2537 Echo("Number of Threes rolled: ", numberOfThreesRolled);
2538 Echo("Number of Fours rolled: ", numberOfFoursRolled);
2539 Echo("Number of Fives rolled: ", numberOfFivesRolled);
2540 Echo("Number of Sixes rolled: ", numberOfSixesRolled);
2541 Echo("Number of Sevens rolled: ", numberOfSevensRolled);
2542 Echo("Number of Eights rolled: ", numberOfEightsRolled);
2543 Echo("Number of Nines rolled: ", numberOfNinesRolled);
2544 Echo("Number of Tens rolled: ", numberOfTensRolled);
2545 Echo("Number of Elevens rolled: ", numberOfElevensRolled);
2546 Echo("Number of Twelves rolled: ", numberOfTwelvesRolled);

2547 %/mathpiper

2548     %output,preserve="false"
2549     Result: True
2550
2551     Side effects:
2552     These are the rolls:
2553     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,
2554     12,7,8,12,6,8,10,10,5,9,8,4,5,3,5,7,7,4,6,7,7,5,8,
2555
2556     Number of Twos rolled: 0
2557     Number of Threes rolled: 3
2558     Number of Fours rolled: 6
2559     Number of Fives rolled: 4
2560     Number of Sixes rolled: 6
2561     Number of Sevens rolled: 13
2562     Number of Eights rolled: 6
2563     Number of Nines rolled: 3
2564     Number of Tens rolled: 2
2565     Number of Elevens rolled: 4
2566     Number of Twelves rolled: 3
2567 .    %/output

```

## 2568 13.6 Exercises

2569 For the following exercises, create a new MathPiperIDE worksheet file called  
 2570 **book\_1\_section\_13\_exercises\_<your first name>\_<your last name>.mpws.**  
 2571 (**Note: there are no spaces in this file name**). For example, John Smith's  
 2572 worksheet would be called:

2573 **book\_1\_section\_13\_exercises\_john\_smith.mpws.**

2574 After this worksheet has been created, place your answer for each exercise that  
2575 requires a fold into its own fold in this worksheet. Place a title attribute in the  
2576 start tag of each fold that indicates the exercise the fold contains the solution to.  
2577 The folds you create should look similar to this one:

2578 `%mathpiper,title="Exercise 1"`

2579 `//Sample fold.`

2580 `%/mathpiper`

2581 If an exercise uses the MathPiper console instead of a fold, copy the work you  
2582 did in the console into a text file so it can be saved.

### 2583 **13.6.1 Exercise 1**

2584 Create a program that uses a While loop to print the even integers from 2  
2585 to 50 inclusive.

### 2586 **13.6.2 Exercise 2**

2587 Create a program that prints all the multiples of 5 between 5 and 50  
2588 inclusive.

### 2589 **13.6.3 Exercise 3**

2590 Create a program that simulates the flipping of a **single coin** 500 times.  
2591 Print the number of times the coin came up heads and the number of times it  
2592 came up tails after the loop is finished executing.



## 2593 14 Predicate Procedures

2594 A **predicate procedure** is a procedure that either returns **True** or **False**. Most  
2595 predicate procedures in MathPiper have names that end with a question mark  
2596 "?". For example, **Even?()**, **Odd?()**, **Integer?()**, etc. The following examples  
2597 show some of the predicate procedures that are in MathPiper:

```
2598 In> Even?(4)
2599 Result: True
```

```
2600 In> Even?(5)
2601 Result: False
```

```
2602 In> Zero?(0)
2603 Result: True
```

```
2604 In> Zero?(1)
2605 Result: False
```

```
2606 In> NegativeInteger?(-1)
2607 Result: True
```

```
2608 In> NegativeInteger?(1)
2609 Result: False
```

```
2610 In> Prime?(7)
2611 Result: True
```

```
2612 In> Prime?(100)
2613 Result: False
```

2614 There is also an **Assigned?()** predicate procedure that can be used to determine  
2615 whether or not a value is assigned to a given variable:

```
2616 In> State()
2617 Result: []
```

```
2618 In> Assigned?(a)
2619 Result: False
```

```
2620 In> a := 1
2621 Result: 1
```

```
2622 In> Assigned?(a)
2623 Result: True
```

```
2624 In> Unassign(a)
2625 Result: True
```

```
2626 In> State
2627 Result: []
```

```
2628 In> Assigned?(a)
2629 Result: False
```

2630 The complete list of predicate procedures is contained in the **Programming**  
2631 **Procedures/Predicates** node in the MathPiperDocs plugin.

### 2632 **14.1 Finding Prime Numbers With A Loop**

2633 Predicate procedures are very powerful when they are combined with loops  
2634 because they can be used to automatically make numerous checks. The  
2635 following program uses a While loop to pass the integers 1 through 20 (one at a  
2636 time) to the **Prime?()** procedure in order to determine which integers are prime  
2637 and which integers are not prime:

```
2638 %mathpiper
```

```
2639 // Determine which integers between 1 and 20 (inclusive)
2640 // are prime and which ones are not prime.
```

```
2641 primes := [];
```

```
2642 nonPrimes := [];
```

```
2643 x := 1;
```

```
2644 while(x <=? 20)
2645 {
2646     primeStatus := Prime?(x);
2647
2648     If(primeStatus ==? True)
2649     {
2650         Append!(primes, x);
2651     }
2652     Else
2653     {
2654         Append!(nonPrimes, x);
2655     }
2656     x := (x + 1);
2657 }
2658 }
```

```
2659 [primes, nonPrimes];
```

```
2660 %/mathpiper
```

```
2661 %output
```

```
2662 Result: [[2,3,5,7,11,13,17,19],[1,4,6,8,9,10,12,14,15,16,18,20]]
```

```
2663 .    %/output
```

2664 This program worked fairly well, but it can be shortened by moving the **Prime?()**  
2665 procedure **inside** of the **If()** procedure instead of using the **primeStatus**  
2666 variable to communicate with it:

```
2667 %mathpiper
```

```
2668 // Determine which integers between 1 and 20 (inclusive)  
2669 // are prime and which ones are not prime.
```

```
2670 primes := [];
```

```
2671 notPrimes := [];
```

```
2672 x := 1;
```

```
2673 While(x <=? 20)  
2674 {  
2675     If(Prime?(x) ==? True)  
2676     {  
2677         Append!(primes, x);  
2678     }  
2679     Else  
2680     {  
2681         Append!(notPrimes, x);  
2682     }  
2683     x := (x + 1);  
2684 }  
2685
```

```
2686 [primes, notPrimes];
```

```
2687 %/mathpiper
```

```
2688     %output  
2689     Result: [[2, 3, 5, 7, 11, 13, 17, 19], [1, 4, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20]]  
2690 .    %/output
```

## 2691 **14.2 Finding The Length Of A String With The Length() Procedure**

2692 Strings can contain zero or more characters, and the **Length()** procedure can be  
2693 used to determine how many characters a string holds:

```
2694 In> s := "Red"  
2695 Result: "Red"
```

```
2696 In> Length(s)  
2697 Result: 3
```

2698 In this example, the string "Red" is assigned to the variable **s** and then **s** is  
2699 passed to the **Length()** procedure. The **Length()** procedure returned a **3**, which  
2700 means the string contained **3 characters**.

2701 The following example shows that strings can also be passed to procedures  
2702 directly:

```
2703 In> Length("Red")  
2704 Result: 3
```

2705 An **empty string** is represented by **two double quote marks with no space in**  
2706 **between them**. The **length** of an empty string is **0**:

```
2707 In> Length("")  
2708 Result: 0
```

### 2709 **14.3 Converting Numbers To Strings With The ToString() Procedure**

2710 Sometimes it is useful to convert a number to a string so that the individual  
2711 digits in the number can be analyzed or manipulated. The following example  
2712 shows a **number** being converted to a **string** with the **ToString()** procedure so  
2713 that its **leftmost** and **rightmost** digits can be assigned to **variables**:

```
2714 In> number := 678  
2715 Result: 678
```

```
2716 In> stringNumber := ToString(number)  
2717 Result: "678"
```

```
2718 In> leftmostDigit := stringNumber[1]  
2719 Result: "6"
```

```
2720 In> rightmostDigit := stringNumber[ Length(stringNumber) ]  
2721 Result: "8"
```

2722 Notice that the Length() procedure is used here to determine which character in  
2723 **stringNumber** held the **rightmost** digit. Also, keep in mind that when numbers  
2724 are in string form, operations such as +, -, \*, and / **cannot** be performed on  
2725 them.

### 2726 **14.4 Finding Prime Numbers that End With 7 (And Multi-line Procedure** 2727 **Calls)**

2728 Now that we have covered how to turn a number into a string, let's use this  
2729 ability inside a loop. The following program finds all the **prime numbers**  
2730 between **1** and **500** that have a **7 as their rightmost digit**. Notice that it has

2731 one If() procedure placed inside of another If() procedure. Placing an If()  
2732 procedure inside of another If() procedure is called **nesting**, and nesting is used  
2733 to to make more complex decisions.

2734 When the program is executed, it finds 24 prime numbers that have 7 as their  
2735 rightmost digit:

```
2736 %mathpiper

2737 /*
2738     Find all the prime numbers between 1 and 500 that have a 7
2739     as their rightmost digit.
2740 */

2741 resultList := [];

2742 x := 1;

2743 While(x <=? 500)
2744 {
2745     If(Prime?(x))
2746     {
2747         stringVersionOfNumber := ToString(x);
2748
2749         stringLength := Length(stringVersionOfNumber);
2750
2751         //Notice that If() procedures can be placed inside of other
2752         // If() procedures.
2753         If(stringVersionOfNumber[stringLength] ==? "7")
2754         {
2755             Append!(resultList, x);
2756         }
2757     }
2758 }
2759 x := (x + 1);
2760 }

2761 resultList;

2762 %/mathpiper

2763
2764 %output
2765 Result: [7, 17, 37, 47, 67, 97, 107, 127, 137, 157, 167, 197, 227,
2766         257, 277, 307, 317, 337, 347, 367, 397, 457, 467, 487]
2767 . %/output
```

## 2768 **14.5 Exercises**

2769 For the following exercises, create a new MathPiperIDE worksheet file called  
2770 **book\_1\_section\_14\_exercises\_<your first name>\_<your last name>.mpws.**  
2771 **(Note: there are no spaces in this file name).** For example, John Smith's  
2772 worksheet would be called:

2773 **book\_1\_section\_14\_exercises\_john\_smith.mpws.**

2774 After this worksheet has been created, place your answer for each exercise that  
2775 requires a fold into its own fold in this worksheet. Place a title attribute in the  
2776 start tag of each fold that indicates the exercise the fold contains the solution to.  
2777 The folds you create should look similar to this one:

2778 `%mathpiper,title="Exercise 1"`

2779 `//Sample fold.`

2780 `%/mathpiper`

2781 If an exercise uses the MathPiper console instead of a fold, copy the work you  
2782 did in the console into a text file so it can be saved.

### 2783 **14.5.1 Exercise 1**

2784 Write a program that uses a While loop to determine **how many** prime numbers  
2785 there are between 1 and 1000. **Do not print the numbers themselves, just**  
2786 **how many there are.**

### 2787 **14.5.2 Exercise 2**

2788 Write a program that uses a While loop to print only the prime numbers  
2789 between 10 and 99 that contain the digit 3 in **either** their ones place **or**  
2790 their tens place.

## 2791 15 More Applications Of Using While Loops With Lists

### 2792 15.1 Adding 1 To Each Element In A List

2793 Procedures that loop can be used to **select each expression in a list in turn** so  
2794 that an operation can be performed on these expressions. The following  
2795 program uses a While loop to select each of the elements in an input list and  
2796 return an output list that contains each of the elements in the input list increased  
2797 by 1:

```
2798 %mathpiper
2799 // Add 1 to each element of a list.
2800 list := [55,93,40,21,7,24,15,14,82];
2801 listLength := Length(list);
2802 index := 1;
2803 while(index <=? listLength)
2804 {
2805     list[index] := (list[index] + 1);
2806     index := (index + 1);
2807 }
2808 list;
2809 %/mathpiper
2810     %output
2811     Result: [56,94,41,22,8,25,16,15,83]
2812 .    %/output
```

### 2813 15.2 Determining If A Number Is In A List

2814 A **loop** can also be used to search through a list. The following program uses a  
2815 **While()** and an **If()** to search through a list to see if it contains the number **53**.  
2816 A message in a string is returned that indicates whether or not 53 was found in  
2817 the list:

```
2818 %mathpiper
2819 //Determine if 53 is in the list.
```

```
2820 testList := [18,26,32,42,53,43,54,6,97,41];
2821 listLength := Length(testList);
2822 result := "53 was not found in the list";
2823 index := 1;
2824 While(index <=? listLength)
2825 {
2826     If(testList[index] == 53)
2827     {
2828         result := "53 was found in the list at position " +
2829             ToString(index);
2830     }
2831     index := (index + 1);
2832 }
2833
2834 result;
2835 %/mathpiper
2836     %output
2837     Result: "53 was found in the list at position 5"
2838 . %/output
```

2839 When this program was executed, it determined that **53** was present in the list at  
2840 position **5**.

### 2841 ***15.3 Finding The Sum Of The Integers In A List Using A While Loop***

```
2842 %mathpiper
2843 // Find the sum all all the integers in a list.
2844 list := [5,10,8,1,6,4,7,7,15,2];
2845 listLength := Length(list);
2846 sum := 0;
2847 index := 1;
2848 While(index <=? listLength)
2849 {
2850     sum := (sum + list[index]);
2851     index := index + 1;
2852 }
2853
```



```
2854 sum;
2855 %/mathpiper
2856     %output
2857     Result: 65
2858 .    %/output
```

## 2859 **15.4 Exercises**

2860 For the following exercises, create a new MathPiperIDE worksheet file called  
2861 **book\_1\_section\_15a\_exercises\_<your first name>\_<your last**  
2862 **name>.mpws**. (**Note: there are no spaces in this file name**). For example,  
2863 John Smith's worksheet would be called:

2864 **book\_1\_section\_15a\_exercises\_john\_smith.mpws**.

2865 After this worksheet has been created, place your answer for each exercise that  
2866 requires a fold into its own fold in this worksheet. Place a title attribute in the  
2867 start tag of each fold that indicates the exercise the fold contains the solution to.  
2868 The folds you create should look similar to this one:

```
2869 %mathpiper,title="Exercise 1"
2870 //Sample fold.
2871 %/mathpiper
```

2872 If an exercise uses the MathPiper console instead of a fold, copy the work you  
2873 did in the console into a text file so it can be saved.

### 2874 **15.4.1 Exercise 1**

2875 Create a program that uses a While loop and the Odd?() predicate procedure  
2876 to analyze the following list and then print the **number** of odd numbers it  
2877 contains. **Hint: think about using code similar to count := (count + 1) in**  
2878 **order to do the counting.**

2879 [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]

### 2880 **15.4.2 Exercise 2**

2881 Create a program that uses a While loop and a NegativeNumber?() procedure  
2882 to **copy** all of the negative numbers in the following list into a **new list**.  
2883 Use the variable **negativeNumbersList** to hold the new list. Print the  
2884 contents of the list after it has been created.

2885 [36, -29, -33, -6, 14, 7, -16, -3, -14, 37, -38, -8, -45, -21, -26, 6, 6, 38, -20, 33, 41, -  
2886 4, 24, 37, 40, 29]

2887 **15.4.3 Exercise 3**

2888 Create one program that uses a single While loop to analyze this list:

2889 [73,12,80,37,56,94,40,21,7,24,15,14,82,93,32,74,22,68,65,52,85,61,46,86,25]

2890 and then print the following information about it:

2891 1) The largest number in the list.

2892 2) The smallest number in the list.

2893 3) The sum of all the numbers in the list (**do not use the Sum() procedure**).

2894 Hint: the following program finds the largest number in a list and it can  
2895 be used as a starting point for solving this exercise.

2896 %mathpiper

2897 /\*

2898 The variable that keeps track of the largest number encountered so

2899 far needs to be initialized to the lowest possible value it may

2900 hold. Why?

2901 \*/

2902 largest := 0;

2903 numbersList := [4,6,2,9,7,1,3];

2904 index := 1;

2905 While(index <=? Length(numbersList) )

2906 {

2907 Echo("Largest: ", largest);

2908

2909 If(numbersList[index] >? largest)

2910 {

2911 largest := numbersList[index]);

2912 }

2913

2914 index := (index + 1);

2915 }

2916 Echo("The largest number in the list is: ", largest);

2917 %/mathpiper

## 2918 **15.5 The ForEach() Looping Procedure**

2919 The **ForEach()** procedure uses a **loop** to index through a list like the While()  
2920 procedure does, but it is more flexible and automatic. ForEach() also uses  
2921 bodied notation like the While() procedure and here is its calling format:

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

2922 **ForEach()** selects each expression in a list in turn, assigns it to the passed-in  
2923 variable, and then executes the expressions that are inside of the body.  
2924 Therefore, body is **executed once for each expression in the list**.

## 2925 **15.6 Print All The Values In A List Using A ForEach() procedure**

2926 This example shows how ForEach() can be used to print all of the items in a list:

```
2927 %mathpiper
2928 //Print all values in a list.
2929 ForEach(value, [50,51,52,53,54,55,56,57,58,59])
2930 {
2931     Echo(value);
2932 }
2933 %/mathpiper
2934 %output,preserve="false"
2935 Result: True
2936
2937 Side Effects:
2938 50
2939 51
2940 52
2941 53
2942 54
2943 55
2944 56
2945 57
2946 58
2947 59
2948 . %/output
```

## 2949 **15.7 Calculate The Sum Of The Numbers In A List Using ForEach()**

2950 In previous examples, counting code in the form **x := (x + 1)** was used to count

2951 how many times a While loop was executed. The following program uses a  
2952 **ForEach()** procedure and a line of code similar to this counter to calculate the  
2953 **sum of the numbers in a list**:

```
2954 %mathpiper
2955 /*
2956    This program calculates the sum of the numbers
2957    in a list.
2958 */
2959 //This variable is used to accumulate the sum.
2960 numbersSum := 0;
2961 ForEach(number, [1,2,3,4,5,6,7,8,9,10] )
2962 {
2963     /*
2964        Add the contents of x to the contents of sum
2965        and place the result back into sum.
2966     */
2967     numbersSum := (numbersSum + number);
2968
2969     //Print the sum as it is being accumulated.
2970     Write(numbersSum, ', ');
2971 }
2972 NewLine(); NewLine();
2973 Echo("The sum of the numbers in the list is ", numbersSum);
2974 %/mathpiper
2975 %output,preserve="false"
2976 Result: True
2977
2978 Side Effects:
2979 1, 3, 6, 10, 15, 21, 28, 36, 45, 55,
2980
2981 The sum of the numbers in the list is 55
2982 . %/output
```

2983 In the above program, the integers **1** through **10** were manually placed into a list  
2984 by typing them individually. This method is limited because only a relatively  
2985 small number of integers can be placed into a list this way. The following section  
2986 discusses an operator that can be used to automatically place a large number of  
2987 integers into a list with very little typing.

## 2988 **15.8 The .. Range Operator**

first .. last

2989 A programmer often needs to create a list that contains **consecutive integers**  
2990 and the .. **"range"** operator can be used to do this. The **first** integer in the list is  
2991 placed before the .. operator and the **last** integer in the list is placed after it  
2992 (**Note: there must be a space immediately to the left of the .. operator**  
2993 **and a space immediately to the right of it or an error will be generated.**).  
2994 Here are some examples:

2995 In> 1 .. 10  
2996 Result: [1,2,3,4,5,6,7,8,9,10]

2997 In> 10 .. 1  
2998 Result: [10,9,8,7,6,5,4,3,2,1]

2999 In> 1 .. 100  
3000 Result: [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,  
3001 21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,  
3002 38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,  
3003 55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,  
3004 72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,  
3005 89,90,91,92,93,94,95,96,97,98,99,100]

3006 In> -10 .. 10  
3007 Result: [-10,-9,-8,-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,8,9,10]

3008 As these examples show, the .. operator can generate lists of integers in  
3009 ascending order and descending order. It can also generate lists that are very  
3010 large and ones that contain negative integers.

3011 Remember, though, if one or both of the spaces around the .. are omitted, an  
3012 error is generated:

3013 In> 1..3  
3014 Result:  
3015 Error parsing expression, near token .3.

## 3016 **15.9 Using ForEach() With The Range Operator To Print The Prime** 3017 **Numbers Between 1 And 100**

3018 The following program shows how to use a **ForEach()** procedure instead of a  
3019 **While()** procedure to print the prime numbers between 1 and 100. Notice that  
3020 loops that are implemented with **ForEach()** often require less typing than  
3021 their **While()** based equivalents:

3022 %mathpiper

```
3023  /*
3024     This program prints the prime integers between 1 and 100 using
3025     a ForEach() procedure instead of a While() procedure. Notice that
3026     the ForEach() version requires less typing than the While()
3027     version.
3028  */

3029  ForEach(number, 1 .. 100)
3030  {
3031      If(Prime?(number)) Write(number, ', ');
3032  }

3033  %/mathpiper

3034      %output,preserve="false"
3035      Result: True
3036
3037      Side Effects:
3038      2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71,
3039      73, 79, 83, 89, 97,
3040  .    %/output
```

### 3041 15.9.1 Using ForEach() And The Range Operator To Place The Prime 3042 Numbers Between 1 And 50 Into A List

3043 A ForEach() procedure can also be used to place values in a list, just like the  
3044 While() procedure can:

```
3045 %mathpiper

3046  /*
3047     Place the prime numbers between 1 and 50 into
3048     a list using a ForEach() procedure.
3049  */

3050  //Create a new list.
3051  primesList := [];

3052  ForEach(number, 1 .. 50)
3053  {
3054      /*
3055         If number is prime, append it to the end of the list and
3056         then assign the new list that is created to the variable
3057         'primes'.
3058      */
3059      If(Prime?(number))
3060      {
3061          primesList := Append(primesList, number);
3062      }
```

```
3063 }
3064 //Print information about the primes that were found.
3065 WriteString("Primes: ");
3066 Write(primesList);
3067 NewLine();
3068 Echo("The number of primes in the list is ", Length(primesList) );
3069 Echo("The first number in the list is ", primesList[1] );
3070 %/mathpiper
3071     %output,preserve="false"
3072     Result: True
3073
3074     Side Effects:
3075     Primes: [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47]
3076     The number of primes in the list is 15
3077     The first number in the list is 2
3078 . %/output
```

3079 As can be seen from the above examples, the **ForEach()** procedure and the  
3080 **range operator** can do a significant amount of work with very little typing. You  
3081 will discover in the next section that MathPiper has procedures that are even  
3082 more powerful than these two.

## 3083 15.9.2 Exercises

3084 For the following exercises, create a new MathPiperIDE worksheet file called  
3085 **book\_1\_section\_15b\_exercises\_<your first name>\_<your last**  
3086 **name>.mpws**. (**Note: there are no spaces in this file name**). For example,  
3087 John Smith's worksheet would be called:

3088 **book\_1\_section\_15b\_exercises\_john\_smith.mpws**.

3089 After this worksheet has been created, place your answer for each exercise that  
3090 requires a fold into its own fold in this worksheet. Place a title attribute in the  
3091 start tag of each fold that indicates the exercise the fold contains the solution to.  
3092 The folds you create should look similar to this one:

```
3093 %mathpiper,title="Exercise 1"
3094 //Sample fold.
3095 %/mathpiper
```

3096 If an exercise uses the MathPiper console instead of a fold, copy the work you  
3097 did in the console into a text file so it can be saved.

**3098 15.9.3 Exercise 1**

3099 Create a program that uses a **ForEach()** procedure and an **Odd?()** predicate  
3100 procedure to analyze the following list and then print the **number** of odd  
3101 numbers it contains.

3102 [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]

**3103 15.9.4 Exercise 2**

3104 Create a program that uses a **ForEach()** procedure and an **NegativeNumber?()**  
3105 procedure to copy all of the negative numbers in the following list into a  
3106 **new list**. Use the variable **negativeNumbersList** to hold the new list.  
3107 Print the contents of the list after it has been created.

3108 [36, -29, -33, -6, 14, 7, -16, -3, -14, 37, -38, -8, -45, -21, -26, 6, 6, 38, -20, 33, 41, -  
3109 4, 24, 37, 40, 29]

**3110 15.9.5 Exercise 3**

3111 Create one program that uses a single **ForEach()** procedure to analyze the  
3112 following list and then print the following information about it:

- 3113 1) The largest number in the list.  
3114 2) The smallest number in the list.  
3115 3) The sum of all the numbers in the list (**do not use the Sum() procedure**).

3116 [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]

**3117 15.9.6 Exercise 4**

3118 Create one program that does the following:

- 3119 1) Use a **While loop** to make a list that contains **1000 random integers**  
3120 between **1** and **100** inclusive.

- 3121 2) Use a **ForEach()** loop to determine **how many** integers in the list you  
3122 created are **prime** and use an **Echo()** procedure to print this total.



## 3123 **16 Procedures & Operators That Loop Internally**

3124 Looping is such a useful capability that MathPiper has many procedures that  
3125 loop internally. Now that you have some experience with loops, you can use this  
3126 experience to help you imagine how these procedures use loops to process the  
3127 information that is passed to them.

### 3128 **16.1 Procedures & Operators That Loop Internally To Process Lists**

3129 This section discusses a number of procedures that use loops to process lists.

#### 3130 **16.1.1 TableForm()**

```
TableForm(list)
```

3131 The **TableForm()** procedure prints the contents of a list in the form of a table.  
3132 Each member in the list is printed on its own line, and this sometimes makes the  
3133 contents of the list easier to read:

```
3134 In> testList := [2,4,6,8,10,12,14,16,18,20]  
3135 Result: [2,4,6,8,10,12,14,16,18,20]
```

```
3136 In> TableForm(testList)  
3137 Result: True  
3138 Side Effects>  
3139 2  
3140 4  
3141 6  
3142 8  
3143 10  
3144 12  
3145 14  
3146 16  
3147 18  
3148 20
```

#### 3149 **16.1.2 Contains?()**

3150 The **Contains?()** procedure searches a list to determine if it contains a given  
3151 expression. If it finds the expression, it returns **True** and if it doesn't find the  
3152 expression, it returns **False**. Here is the calling format for Contains?():

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

3153 The following code shows Contains?() being used to locate a number in a list:

3154 In> Contains?([50,51,52,53,54,55,56,57,58,59], 53)  
3155 Result: True

3156 In> Contains?([50,51,52,53,54,55,56,57,58,59], 75)  
3157 Result: False

3158 The !? operator can also be used with predicate procedures like Contains?() to  
3159 change their results to the opposite truth value:

3160 In> !? Contains?([50,51,52,53,54,55,56,57,58,59], 75)  
3161 Result: True

### 3162 16.1.3 Find()

Find(list, expression)

3163 The **Find()** procedure searches a list for the first occurrence of a given  
3164 expression. If the expression is found, the **position of its first occurrence** is  
3165 returned and if it is not found, **-1** is returned:

3166 In> Find([23, 15, 67, 98, 64], 15)  
3167 Result: 2

3168 In> Find([23, 15, 67, 98, 64], 8)  
3169 Result: -1

### 3170 16.1.4 Count()

Count(list, expression)

3171 **Count()** determines the number of times a given expression occurs in a list:

3172 In> testList := [\_a,\_b,\_b,\_c,\_c,\_c,\_d,\_d,\_d,\_d,\_e,\_e,\_e,\_e,\_e]  
3173 Result: [\_a,\_b,\_b,\_c,\_c,\_c,\_d,\_d,\_d,\_d,\_e,\_e,\_e,\_e,\_e]

3174 In> Count(testList, \_c)  
3175 Result: 3

3176 In> Count(testList, \_e)  
3177 Result: 5

3178 In> Count(testList, \_z)  
3179 Result: 0

3180 **16.1.5 Select()**

```
Select(list, predicate_procedure)
```

3181 **Select()** returns a list that contains all the expressions in a list that make a given  
3182 predicate procedure return **True**:

```
3183 In> Select([46,87,59,-27,11,86,-21,-58,-86,-52], "PositiveInteger?")  
3184 Result: [46,87,59,11,86]
```

3185 In this example, notice that the **name** of the predicate procedure is passed to  
3186 Select() in **double quotes**. There are other ways to pass a predicate procedure  
3187 to Select() but these are covered in a later section.

3188 Here are some further examples that use the Select() procedure:

```
3189 In> Select([16,14,82,92,33,74,99,67,65,52], "Odd?")  
3190 Result: [33,99,67,65]
```

```
3191 In> Select([16,14,82,92,33,74,99,67,65,52], "Even?")  
3192 Result: [16,14,82,92,74,52]
```

```
3193 In> Select(1 .. 75, "Prime?")  
3194 Result: [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73]
```

3195 Notice how the third example uses the **..** operator to automatically generate a list  
3196 of consecutive integers from 1 to 75 for the Select() procedure to analyze.

3197 **16.1.6 The Nth() Procedure & The [] Operator**

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

3198 The **Nth()** procedure simply returns the expression that is at a given position in  
3199 a list. This example shows the **third** expression in a list being obtained:

```
3200 In> testList := [_a,_b,_c,_d,_e,_f,_g]  
3201 Result: [_a,_b,_c,_d,_e,_f,_g]
```

```
3202 In> Nth(testList, 3)  
3203 Result: c
```

3204 As discussed earlier, the **[]** operator can also be used to obtain a single  
3205 expression from a list:

3206 In> testList[3]  
3207 Result: c

3208 The **[]** operator can even obtain a single expression directly from a list without  
3209 needing to use a variable:

3210 In> [\_a,\_b,\_c,\_d,\_e,\_f,\_g][3]  
3211 Result: \_c

### 3212 16.1.7 Concat()

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

3213 The Concat() procedure is short for "concatenate", which means to join together  
3214 sequentially. It takes two or more lists and joins them together into a single  
3215 larger list:

3216 In> Concat([\_a,\_b,\_c], [1,2,3], [\_x,\_y,\_z])  
3217 Result: [\_a,\_b,\_c,1,2,3,\_x,\_y,\_z]

### 3218 16.1.8 Insert(), Delete(), & Replace()

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

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

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

3219 **Insert()** inserts an expression into a list at a given index, **Delete()** deletes an  
3220 expression from a list at a given index, and **Replace()** replaces an expression in  
3221 a list at a given index with another expression:

3222 In> testList := [\_a,\_b,\_c,\_d,\_e,\_f,\_g]  
3223 Result: [\_a,\_b,\_c,\_d,\_e,\_f,\_g]

3224 In> testList := Insert(testList, 4, 123)  
3225 Result: [\_a,\_b,\_c,123,\_d,\_e,\_f,\_g]

3226 In> testList := Delete(testList, 4)  
3227 Result: [\_a,\_b,\_c,\_d,\_e,\_f,\_g]

3228 In> testList := Replace(testList, 4, \_xxx)

3229 Result: [\_a,\_b,\_c,\_xxx,\_e,\_f,\_g]

### 3230 16.1.9 Take()

```
Take(list, amount)
Take(list, -amount)
Take(list, [begin_index,end_index])
```

3231 **Take()** obtains a sublist from the **beginning** of a list, the **end** of a list, or the  
3232 **middle** of a list. The expressions in the list that are not taken are discarded.

3233 A **positive** integer passed to Take() indicates how many expressions should be  
3234 taken from the **beginning** of a list:

3235 In> testList := [\_a,\_b,\_c,\_d,\_e,\_f,\_g]  
3236 Result: [\_a,\_b,\_c,\_d,\_e,\_f,\_g]

3237 In> Take(testList, 3)  
3238 Result: [\_a,\_b,\_c]

3239 A **negative** integer passed to Take() indicates how many expressions should be  
3240 taken from the **end** of a list:

3241 In> Take(testList, -3)  
3242 Result: [\_e,\_f,\_g]

3243 Finally, if a **two member list** is passed to Take() it indicates the **range** of  
3244 expressions that should be taken from the **middle** of a list. The **first** value in the  
3245 passed-in list specifies the **beginning** index of the range and the **second** value  
3246 specifies its **end**:

3247 In> Take(testList, [3,5])  
3248 Result: [\_c,\_d,\_e]

### 3249 16.1.10 Drop()

```
Drop(list, index)
Drop(list, -index)
Drop(list, [begin_index,end_index])
```

3250 **Drop()** does the opposite of Take() in that it **drops** expressions from the  
3251 **beginning** of a list, the **end** of a list, or the **middle** of a list, and **returns a list**  
3252 **that contains the remaining expressions.**

3253 A **positive** integer passed to Drop() indicates how many expressions should be

3254 dropped from the **beginning** of a list:

3255 In> testList := [\_a,\_b,\_c,\_d,\_e,\_f,\_g]

3256 Result: [\_a,\_b,\_c,\_d,\_e,\_f,\_g]

3257 In> Drop(testList, 3)

3258 Result: [\_d,\_e,\_f,\_g]

3259 A **negative** integer passed to Drop() indicates how many expressions should be  
3260 dropped from the **end** of a list:

3261 In> Drop(testList, -3)

3262 Result: [\_a,\_b,\_c,\_d]

3263 Finally, if a **two member list** is passed to Drop() it indicates the **range** of  
3264 expressions that should be dropped from the **middle** of a list. The **first** value in  
3265 the passed-in list specifies the **beginning** index of the range and the **second**  
3266 value specifies its **end**:

3267 In> Drop(testList, [3,5])

3268 Result: [\_a,\_b,\_f,\_g]

### 3269 16.1.11 FillList()

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

3270 The FillList() procedure simply creates a list that is of size "length" and fills it  
3271 with "length" copies of the given expression:

3272 In> FillList(\_a, 5)

3273 Result: [\_a,\_a,\_a,\_a,\_a]

3274 In> FillList(42,8)

3275 Result: [42,42,42,42,42,42,42,42]

### 3276 16.1.12 RemoveDuplicates()

```
RemoveDuplicates(list)
```

3277 **RemoveDuplicates()** removes any duplicate expressions that are contained in a  
3278 list:

3279 In> testList := [\_a,\_a,\_b,\_c,\_c,\_b,\_b,\_a,\_b,\_c,\_c]

3280 Result: [\_a,\_a,\_b,\_c,\_c,\_b,\_b,\_a,\_b,\_c,\_c]

```
3281 In> RemoveDuplicates(testList)
3282 Result: [_a,_b,_c]
```

### 3283 16.1.13 Reverse()

```
Reverse(list)
```

3284 **Reverse()** reverses the order of the expressions in a list:

```
3285 In> testList := [_a,_b,_c,_d,_e,_f,_g,_h]
3286 Result: [_a,_b,_c,_d,_e,_f,_g,_h]
```

```
3287 In> Reverse(testList)
3288 Result: [_h,_g,_f,_e,_d,_c,_b,_a]
```

### 3289 16.1.14 Partition()

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

3290 The **Partition()** procedure breaks a list into sublists of size "partition\_size":

```
3291 In> testList := [_a,_b,_c,_d,_e,_f,_g,_h]
3292 Result: [_a,_b,_c,_d,_e,_f,_g,_h]
```

```
3293 In> Partition(testList, 2)
3294 Result: [[_a,_b],[_c,_d],[_e,_f],[_g,_h]]
```

3295 If the partition\_size does not divide the length of the list **evenly**, the remaining  
3296 elements are discarded:

```
3297 In> Partition(testList, 3)
3298 Result: [[_a,_b,_c],[_d,_e,_f]]
```

3299 The number of elements that Partition() will discard can be calculated by  
3300 dividing the length of a list by the partition size and obtaining the **remainder**:

```
3301 In> Length(testList) % 3
3302 Result: 2
```

3303 Remember that % is the remainder operator. It divides two integers and returns  
3304 their remainder.

3305 **16.1.15 BuildList()**

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

3306 The BuildList() procedure creates a list of values by doing the following:

- 3307 1) Generating a sequence of values between a "begin\_value" and an  
3308 "end\_value" with each value being incremented by the "step\_amount".
- 3309 2) Placing each value in the sequence into the specified "variable", one value  
3310 at a time.
- 3311 3) Evaluating the defined "expression" (which contains the defined "variable")  
3312 for each value, one at a time.
- 3313 4) Placing the result of each "expression" evaluation into the result list.

3314 This example generates a list that contains the integers 1 through 10:

```
3315 In> BuildList(x, x, 1, 10, 1)
3316 Result: [1,2,3,4,5,6,7,8,9,10]
```

3317 Notice that the expression in this example is simply the variable 'x' itself with no  
3318 other operations performed on it.

3319 The following example is similar to the previous one except that its expression  
3320 multiplies 'x' by 2:

```
3321 In> BuildList(x*2, x, 1, 10, 1)
3322 Result: [2,4,6,8,10,12,14,16,18,20]
```

3323 Lists that contain decimal values can also be created by setting the  
3324 "step\_amount" to a decimal:

```
3325 In> BuildList(x, x, 0, 1, .1)
3326 Result: [0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1]
```

3327 **16.1.16 Sort()**

```
Sort(list, compare)
```

3328 **Sort()** sorts the elements of **list** into the order indicated by **compare** with  
3329 compare typically being the **less than** operator "<" or the **greater than**  
3330 operator ">":

```
3331 In> Sort([4,7,23,53,-2,1], "<?");
3332 Result: [-2,1,4,7,23,53]
```



```
3333 In> Sort([4,7,23,53,-2,1], ">?");
3334 Result: [53,23,7,4,1,-2]
```

```
3335 In> Sort([1/2,3/5,7/8,5/16,3/32], "<?")
3336 Result: [3/32,5/16,1/2,3/5,7/8]
```

```
3337 In> Sort([.5,3/5,.76,5/16,3/32], "<?")
3338 Result: [3/32,5/16,.5,3/5,.76]
```

## 3339 **16.2 Procedures That Work With Integers**

3340 This section discusses various procedures that work with integers. Some of  
3341 these procedures also work with non-integer values and their use with non-  
3342 integers is discussed in other sections.

### 3343 **16.2.1 RandomIntegerList()**

```
RandomIntegerList(length, lowest_possible, highest_possible)
```

3344 A vector is a list that does not contain other lists. **RandomIntegerList()** creates  
3345 a list of size "length" that contains random integers that are no lower than  
3346 "lowest\_possible" and no higher than "highest possible". The following example  
3347 creates **10** random integers between **1** and **99** inclusive:

```
3348 In> RandomIntegerList(10, 1, 99)
3349 Result: [73,93,80,37,55,93,40,21,7,24]
```

### 3350 **16.2.2 Maximum() & Minimum()**

```
Maximum(value1, value2)
Maximum(list)
```

3351 If two values are passed to Maximum(), it determines which one is larger:

```
3352 In> Maximum(10, 20)
3353 Result: 20
```

3354 If a list of values are passed to Maximum(), it finds the largest value in the list:

```
3355 In> testList := RandomIntegerList(10, 1, 99)
3356 Result: [73,93,80,37,55,93,40,21,7,24]
```

```
3357 In> Maximum(testList)
3358 Result: 93
```

3359 The **Minimum()** procedure is the opposite of the Maximum() procedure.

```
Minimum(value1, value2)
Minimum(list)
```

3360 If two values are passed to Minimum(), it determines which one is smaller:

```
3361 In> Minimum(10, 20)
3362 Result: 10
```

3363 If a list of values are passed to Minimum(), it finds the smallest value in the list:

```
3364 In> testList := RandomIntegerList(10, 1, 99)
3365 Result: [73,93,80,37,55,93,40,21,7,24]
```

```
3366 In> Minimum(testList)
3367 Result: 7
```

### 3368 16.2.3 Quotient() & Modulo()

```
Quotient(dividend, divisor)
Modulo(dividend, divisor)
```

3369 **Quotient()** determines the whole number of times a divisor goes into a dividend:

```
3370 In> Quotient(7, 3)
3371 Result: 2
```

3372 **Modulo()** determines the **remainder** that results when a dividend is divided by  
3373 a divisor:

```
3374 In> Modulo(7,3)
3375 Result: 1
```

3376 The remainder/modulo operator % can also be used to calculate a remainder:

```
3377 In> 7 % 2
3378 Result: 1
```

### 3379 16.2.4 Gcd()

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

3380 GCD stands for Greatest Common Divisor and the **Gcd()** procedure determines  
3381 the greatest common divisor of the values that are passed to it.

3382 If two integers are passed to Gcd(), it calculates their greatest common divisor:

3383 In> Gcd(21, 56)

3384 Result: 7

3385 If a list of integers are passed to Gcd(), it finds the greatest common divisor of all  
3386 the integers in the list:

3387 In> Gcd([9, 66, 123])

3388 Result: 3

### 3389 16.2.5 Lcm()

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

3390 LCM stands for Least Common Multiple and the **Lcm()** procedure determines  
3391 the least common multiple of the values that are passed to it.

3392 If two integers are passed to Lcm(), it calculates their least common multiple:

3393 In> Lcm(14, 8)

3394 Result: 56

3395 If a list of integers are passed to Lcm(), it finds the least common multiple of all  
3396 the integers in the list:

3397 In> Lcm([3, 7, 9, 11])

3398 Result: 693

### 3399 16.2.6 Sum()

```
Sum(list)
```

3400 **Sum()** can find the sum of a list that is passed to it:

3401 In> testList := RandomIntegerList(10,1,99)

3402 Result: [73,93,80,37,55,93,40,21,7,24]

3403 In> Sum(testList)

3404 Result: 523

3405 In> testList := (1 .. 10)

3406 **Result:** [1,2,3,4,5,6,7,8,9,10]

3407 In> Sum(testList)

3408 **Result:** 55

### 3409 **16.2.7 Product()**

Product(list)

3410 This procedure has two calling formats, only one of which is discussed here.

3411 Product(**list**) multiplies all the expressions in a list together and returns their  
3412 product:

3413 In> Product([1,2,3])

3414 **Result:** 6

### 3415 **16.3 Exercises**

3416 For the following exercises, create a new MathPiperIDE worksheet file called

3417 **book\_1\_section\_16\_exercises\_<your first name>\_<your last name>.mpws.**

3418 (**Note: there are no spaces in this file name**). For example, John Smith's

3419 worksheet would be called:

3420 **book\_1\_section\_16\_exercises\_john\_smith.mpws.**

3421 After this worksheet has been created, place your answer for each exercise that  
3422 requires a fold into its own fold in this worksheet. Place a title attribute in the  
3423 start tag of each fold that indicates the exercise the fold contains the solution to.  
3424 The folds you create should look similar to this one:

3425 %mathpiper,title="Exercise 1"

3426 //Sample fold.

3427 %/mathpiper

3428 If an exercise uses the MathPiper console instead of a fold, copy the work you  
3429 did in the console into a text file so it can be saved.

### 3430 **16.3.1 Exercise 1**

3431 Create a program that uses **RandomIntegerList()** to create a 100 member list  
3432 that contains random integers between 1 and 5 inclusive. Use one **Count()**  
3433 procedure call in a loop to determine how many of each digit 1-5 are in the  
3434 list and then print this information.

3435 Hint 1: You can use the following code as the starting point for your loop:

```
3436  ForEach(num, 1 .. 5)
3437  {
3438  }
```

3439 Hint 2: you can use the `Sort()` procedure to sort the generated list to make  
3440 it easier to check if your program is counting correctly.

### 3441 **16.3.2 Exercise 2**

3442 Create a program that uses **RandomIntegerList()** to create a 100 member list  
3443 that contains random integers between 1 and 50 inclusive and use **Contains?**  
3444 **()** to determine if the number 25 is in the list. Print "25 was in the  
3445 list." if 25 was found in the list and "25 was not in the list." if it  
3446 wasn't found.

### 3447 **16.3.3 Exercise 3**

3448 Create a program that uses **RandomIntegerList()** to create a 100 member list  
3449 that contains random integers between 1 and 50 inclusive and use **Find()** to  
3450 determine if the number 10 is in the list. Print the position of 10 if it  
3451 was found in the list and "10 was not in the list." if it wasn't found.

### 3452 **16.3.4 Exercise 4**

3453 Create a program that uses **RandomIntegerList()** to create a 100 member list  
3454 that contains random integers between 0 and 3 inclusive. Use **Select()** with  
3455 the **NonZeroInteger?()** predicate procedure to obtain all of the nonzero  
3456 integers in this list.

### 3457 **16.3.5 Exercise 5**

3458 Create a program that uses **BuildList()** to obtain a list that contains the  
3459 squares of the integers between 1 and 10 inclusive.

## 3460 17 Nested Loops

3461 Now that you have seen how to solve problems with single loops, it is time to  
3462 discuss what can be done when a loop is placed inside of another loop. A loop  
3463 that is placed **inside** of another loop it is called a **nested loop** and this nesting  
3464 can be extended to numerous levels if needed. This means that loop 1 can have  
3465 loop 2 placed inside of it, loop 2 can have loop 3 placed inside of it, loop 3 can  
3466 have loop 4 placed inside of it, and so on.

3467 Nesting loops allows the programmer to accomplish an enormous amount of  
3468 work with very little typing.

### 3469 17.1 Generate All The Combinations That Can Be Entered Into A Two Digit 3470 Wheel Lock Using A Nested Loop



3471 The following program generates all the combinations that can be entered into a  
3472 two digit wheel lock. It uses a nested loop to accomplish this with the "**inside**"  
3473 nested loop being used to generate **one's place** digits and the "**outside**" loop  
3474 being used to generate **ten's place** digits.

```
3475 %mathpiper
```

```
3476 /*  
3477     Generate all the combinations can be entered into a two  
3478     digit wheel lock.  
3479 */
```

```
3480 combinationsList := [];
```

```
3481 ForEach(digit1, 0 .. 9) //This loop is called the "outside" loop.
```

```

3482 {
3483     ForEach(digit2, 0 .. 9)//This loop is called the "inside" loop.
3484     {
3485         combinationsList := Append(combinationsList, [digit1, digit2]);
3486     }
3487 }

3488 TableForm(combinationsList);

3489 %/mathpiper

3490     %output,preserve="false"
3491     Result: True
3492
3493     Side Effects:
3494     [0,0]
3495     [0,1]
3496     [0,2]
3497     [0,3]
3498     [0,4]
3499     [0,5]
3500     [0,6]
3501     .
3502     . //The middle of the list has not been shown.
3503     .
3504     [9,3]
3505     [9,4]
3506     [9,5]
3507     [9,6]
3508     [9,7]
3509     [9,8]
3510     [9,9]
3511     True
3512 . %/output

```

3513 The relationship between the outside loop and the inside loop is interesting  
 3514 because each time the **outside loop cycles once**, the **inside loop cycles 10**  
 3515 **times**. Study this program carefully because nested loops can be used to solve a  
 3516 wide range of problems and therefore understanding how they work is  
 3517 important.

## 3518 17.2 Exercises

3519 For the following exercises, create a new MathPiperIDE worksheet file called  
 3520 **book\_1\_section\_17\_exercises\_<your first name>\_<your last name>.mpws**.  
 3521 (**Note: there are no spaces in this file name**). For example, John Smith's  
 3522 worksheet would be called:  
 3523 **book\_1\_section\_17\_exercises\_john\_smith.mpws**.

3524 After this worksheet has been created, place your answer for each exercise that  
3525 requires a fold into its own fold in this worksheet. Place a title attribute in the  
3526 start tag of each fold that indicates the exercise the fold contains the solution to.  
3527 The folds you create should look similar to this one:

3528 `%mathpiper,title="Exercise 1"`

3529 `//Sample fold.`

3530 `%/mathpiper`

3531 If an exercise uses the MathPiper console instead of a fold, copy the work you  
3532 did in the console into a text file so it can be saved.

### 3533 **17.2.1 Exercise 1**

3534 Create a program that will generate all of the combinations that can be  
3535 entered into a three digit wheel lock. (Hint: a triple nested loop can be  
3536 used to accomplish this.)



## 3537 18 User Defined Procedures

3538 In computer programming, a **procedure** is a named section of code that can be  
3539 **called** from other sections of code. **Values** can be sent to a procedure for  
3540 processing as part of the **call**, and a procedure always returns a value as its  
3541 result. A procedure can also generate side effects when it is called, and side  
3542 effects have been covered in earlier sections.

3543 The values that are sent to a procedure when it is called are called **arguments**  
3544 or **actual parameters**, and a procedure can accept 0 or more of them. These  
3545 arguments are usually placed within parentheses.

3546 MathPiper has many predefined procedures (some of which have been discussed  
3547 in previous sections) but users can create their own procedures too. The  
3548 following program creates a procedure called **addNums()** that takes two  
3549 numbers as arguments, adds them together, and returns their sum back to the  
3550 calling code as a result:

```
3551 In> addNums(num1,num2) := (num1 + num2)
3552 Result: True
```

3553 This line of code defined a new procedure called **addNums** and specified that it  
3554 will accept two values when it is called. The **first** value will be assigned to the  
3555 variable **num1** and the **second** value will be assigned to the variable **num2**.

3556 Variables like num1 and num2 that are used in a procedure to accept values from  
3557 calling code are called **formal parameters**. **Formal parameter variables** are  
3558 used inside a procedure to process the **values/actual parameters/arguments**  
3559 that were assigned to them by the calling code.

3560 The code on the **right side** of the **assignment operator** is **assigned** to the  
3561 procedure name "**addNums**" and it is executed each time **addNums()** is called.  
3562 The following example shows the new **addNums()** procedure being called  
3563 multiple times with different values being passed to it:

```
3564 In> addNums(2,3)
3565 Result: 5
```

```
3566 In> addNums(4,5)
3567 Result: 9
```

```
3568 In> addNums(9,1)
3569 Result: 10
```

3570 Notice that, unlike the procedures that come with MathPiper, we chose to have  
3571 this procedure's name start with a **lower case letter**. We could have had  
3572 addNums() begin with an upper case letter but it is a **convention** in MathPiper

3573 for **user defined procedure names to begin with a lower case letter to**  
3574 **distinguish them from the procedures that come with MathPiper.**

3575 The values that are returned from user defined procedures can also be assigned  
3576 to variables. The following example uses a %mathpiper fold to define a  
3577 procedure called **evenIntegers()** and then this procedure is used in the  
3578 MathPiper console to assign a list of even integers to the variable "a":

```
3579 %mathpiper
3580 evenIntegers(endInteger) :=
3581 {
3582     resultList := [];
3583     x := 2;
3584     while(x <=? endInteger)
3585     {
3586         resultList := Append(resultList, x);
3587         x := (x + 2);
3588     }
3589     /*
3590     The result of the last expression that is executed in a procedure
3591     is the result that the procedure returns to the caller. In this case,
3592     resultList is purposely being executed last so that its contents are
3593     returned to the caller.
3594     */
3595     resultList;
3596 }
3597
3599 %/mathpiper
3600     %output,preserve="false"
3601     Result: True
3602 . %/output
3603 In> a := evenIntegers(10)
3604 Result: [2,4,6,8,10]
3605 In> Length(a)
3606 Result: 5
```

3607 The procedure **evenIntegers()** returns a list that contains all the even integers  
3608 from 2 up through the value that was passed into it. The fold was first executed  
3609 in order to define the **evenIntegers()** procedure and make it ready for use. The  
3610 **evenIntegers()** procedure was then called from the MathPiper console and 10  
3611 was passed to it.

3612 After the procedure was finished executing, it returned a list of even integers as  
3613 a result, and this result was assigned to the variable 'a'. We then passed the list  
3614 that was assigned to 'a' to the **Length()** procedure in order to determine its size.

## 3615 **18.1 Global Variables, Local Variables, & Local()**

3616 The new **evenIntegers()** procedure seems to work well, but there is a problem.  
3617 The variables '**x**' and **resultList** were defined inside the procedure as **global**  
3618 **variables**, which means they are accessible from anywhere, including from  
3619 within other procedures, within other folds (as shown here):

```
3620 %mathpiper
3621 Echo(x, ",", resultList);
3622 %/mathpiper
3623     %output,preserve="false"
3624     Result: True
3625
3626     Side Effects:
3627     12 , [2,4,6,8,10]
3628 . %/output
```

3629 and from within the MathPiper console:

```
3630 In> x
3631 Result: 12
3632 In> resultList
3633 Result: [2,4,6,8,10]
```

3634 **Using global variables inside of procedures is usually not a good idea**  
3635 because code in other procedures and folds might already be using (or will use)  
3636 the same variable names. Global variables that have the same name are the  
3637 same variable. When one section of code changes the value of a given global  
3638 variable, the value is changed everywhere that variable is used and this will  
3639 eventually cause problems.

3640 In order to prevent errors being caused by global variables having the same  
3641 name, a procedure named **Local()** can be called inside of a procedure to define  
3642 what are called **local variables**. A **local variable** is only accessible inside the  
3643 procedure it has been defined in, even if it has the same name as a global  
3644 variable. The following example shows a second version of the **evenIntegers()**  
3645 procedure that uses **Local()** to make '**x**' and **resultList** local variables:

```
3646 %mathpiper
3647 /*
3648  This version of evenIntegers() uses Local() to make
3649  x and resultList local variables
3650 */
3651 evenIntegers(endInteger) :=
3652 {
3653     Local(x,resultList);
3654     resultList := [];
3655
3656     x := 2;
3657     While(x <=? endInteger)
3658     {
3659         resultList := Append(resultList, x);
3660
3661         x := (x + 2);
3662     }
3663
3664     /*
3665     The result of the last expression that is executed in a procedure
3666     is the result that the procedure returns to the caller.  In this case,
3667     resultList is purposely being executed last so that its contents are
3668     returned to the caller.
3669     */
3670     resultList;
3671 }
3672 %/mathpiper
3673     %output,preserve="false"
3674     Result: True
3675 . %/output
```

3676 We can verify that '**x**' and **resultList** are now local variables by first clearing  
3677 them, calling **evenIntegers()**, and then seeing what '**x**' and **resultList** contain:

```
3678 In> Unassign(x, resultList)
3679 Result: True
3680 In> evenIntegers(10)
3681 Result: [2,4,6,8,10]
3682 In> x
3683 Result: x
3684 In> resultList
```

3685 `Result: resultList`

## 3686 **18.2 Exercises**

3687 For the following exercises, create a new MathPiperIDE worksheet file called  
3688 **book\_1\_section\_18\_exercises\_<your first name>\_<your last name>.mpws**.  
3689 (**Note: there are no spaces in this file name**). For example, John Smith's  
3690 worksheet would be called:

3691 **book\_1\_section\_18\_exercises\_john\_smith.mpws**.

3692 After this worksheet has been created, place your answer for each exercise that  
3693 requires a fold into its own fold in this worksheet. Place a title attribute in the  
3694 start tag of each fold that indicates the exercise the fold contains the solution to.  
3695 The folds you create should look similar to this one:

3696 `%mathpiper,title="Exercise 1"`

3697 `//Sample fold.`

3698 `%/mathpiper`

3699 If an exercise uses the MathPiper console instead of a fold, copy the work you  
3700 did in the console into a text file so it can be saved.

### 3701 **18.2.1 Exercise 1**

3702 Create a procedure called **tenOddIntegers()** that returns a list that  
3703 contains 10 random odd integers between 1 and 99 inclusive.

3704 Hint: You may want to use the `RandomIntegerList()`, `Select()`, `Odd?()`, and  
3705 `Take()` procedures.

### 3706 **18.2.2 Exercise 2**

3707 Create a procedure called **convertStringToList(string)** that takes a string  
3708 as a parameter and returns a list that contains all of the characters in  
3709 the string. Here is an example of how the procedure should work:

3710 `In> convertStringToList("Hello friend!")`

3711 `Result: ["H","e","l","l","o"," ","f","r","i","e","n","d","!"]`

3712 `In> convertStringToList("Computer Algebra System")`

3713 `Result: ["C","o","m","p","u","t","e","r"," ","A","l","g","e","b","r","a"," ","S","y","s","t","e","m"]`  
3714

3715 Hint: Remember, a string can be broken down into individual characters by using  
3716 an index value inside of brackets `[]` like this:

```
3717 In> string := "Hello"
3718 Result: "Hello"
```

```
3719 In> string[1]
3720 Result: "H"
```

```
3721 In> string[2]
3722 Result: "e"
```

```
3723 In> string[3]
3724 Result: "l"
```

```
3725 In> string[4]
3726 Result: "l"
```

```
3727 In> string[5]
3728 Result: "o"
```

```
3729 Your procedure should use this indexing technique inside of a loop to append
3730 each of these characters to a list.
```

## 3731 19 Miscellaneous topics

### 3732 19.1 Incrementing And Decrementing Variables With The ++ And -- 3733 Operators

3734 Up until this point we have been adding 1 to a variable with code in the form of **x**  
3735 **:= (x + 1)** and subtracting 1 from a variable with code in the form of **x := (x -**  
3736 **1)**. Another name for **adding** 1 to a variable is **incrementing** it and  
3737 **decrementing** a variable means to **subtract** 1 from it. Now that you have had  
3738 some experience with these longer forms, it is time to show you shorter versions  
3739 of them.

#### 3740 19.1.1 Incrementing Variables With The ++ Operator

3741 The number 1 can be added to a variable by simply placing the ++ operator after  
3742 it like this:

```
3743 In> x := 1  
3744 Result: 1
```

```
3745 In> x++;  
3746 Result: 2
```

```
3747 In> x  
3748 Result: 2
```

3749 Here is a program that uses the ++ operator to increment a loop index variable:

```
3750 %mathpiper
```

```
3751 index := 1;
```

```
3752 while(index <=? 10)  
3753 {  
3754     Echo(index);  
3755     index++; //The ++ operator increments the index variable.  
3756 }  
3757
```

```
3758 %/mathpiper
```

```
3759     %output,preserve="false"  
3760     Result: True  
3761  
3762     Side Effects:  
3763     1
```

```
3764      2
3765      3
3766      4
3767      5
3768      6
3769      7
3770      8
3771      9
3772     10
3773 .    %/output
```

### 3774 19.1.2 Decrementing Variables With The -- Operator

3775 The number 1 can be subtracted from a variable by simply placing the --  
3776 operator after it like this:

```
3777 In> x := 1
3778 Result: 1
```

```
3779 In> x--;
3780 Result: 0
```

```
3781 In> x
3782 Result: 0
```

3783 Here is a program that uses the -- operator to decrement a loop index variable:

```
3784 %mathpiper
```

```
3785 index := 10;
```

```
3786 while(index >= 1)
3787 {
3788     Echo(index);
3789     index--; //The -- operator decrements the index variable.
3790 }
3791 }
```

```
3792 %/mathpiper
```

```
3793 %output,preserve="false"
3794 Result: True
3795
3796 Side Effects:
3797 10
3798 9
3799 8
3800 7
3801 6
```



```
3802     5
3803     4
3804     3
3805     2
3806     1
3807 .    %/output
```

### 3808 19.1.3 The For() Looping Procedure

3809 The For() procedure provides an easy way to create loops that use an index  
3810 variable. This is the calling format for the For() procedure:

```
For(initialization, predicate, changeIndex) body
```

3811 The parameter named "initialization" is an expression that is usually used to  
3812 assign an initial value to the index variable. The parameter named "predicate" is  
3813 an expression that is evaluated before the body is evaluated. If this "predicate"  
3814 evaluates to True, then the body is evaluated. If "predicate" evaluates to False,  
3815 the body is not evaluated, and the For() procedure finishes. The parameter  
3816 named "changeIndex" is used to increase or decrease the value that is assigned  
3817 to the index variable.

3818 The following code uses a For() procedure to print the integers from 1 to 10  
3819 inclusive:

```
3820 %mathpiper
3821 For(index := 1, index <=? 10, index++)
3822 {
3823     Echo(index);
3824 }
```

```
3825 %/mathpiper
3826 %output,preserve="false"
3827 Result: True
3828
3829 Side Effects:
3830 1
3831 2
3832 3
3833 4
3834 5
3835 6
3836 7
3837 8
3838 9
```

```
3839     10
3840 .    %/output
```

#### 3841 19.1.4 The Break() Procedure

3842 The **Break()** procedure is used to end a loop early and here is its calling format:

```
Break()
```

3843 The following program has a While loop that is configured to loop 10 times.  
3844 However, when the loop counter variable **index** reaches 5, the Break() procedure  
3845 is called and this causes the loop to end early:

```
3846 %mathpiper
3847
3848 index := 1;
3849
3850 While(index <=? 10)
3851 {
3852     Echo(index);
3853
3854     If(index ==? 5) Break();
3855
3856     index++;
3857 }
3858
3859 %/mathpiper
3860
3861 %output,preserve="false"
3862     Result: True
3863
3864     Side Effects:
3865     1
3866     2
3867     3
3868     4
3869     5
3869 .    %/output
```

3870 When a Break() procedure is used to end a loop, it is called "**breaking out**" of  
3871 the loop. Notice that only the numbers 1-5 are printed in this program.

#### 3872 19.1.5 The Continue() Procedure

3873 The **Continue()** procedure is similar to the Break() procedure, except that

3874 instead of ending the loop, it simply causes it to **skip the remainder of the**  
3875 **loop for the current loop iteration**. Here is the Continue() procedure's calling  
3876 format:

```
Continue()
```

3877 The following program uses a While loop that is configured to print the integers  
3878 from 0 to 8. However, the Continue() procedure is used to skip the execution of  
3879 the Echo() procedure when the loop indexing variable **index** is equal to 5:

```
3880 %mathpiper
3881
3882 index := 0;
3883
3884 While(index <? 8)
3885 {
3886     index++;
3887
3888     If(index ==? 5) Continue();
3889
3890     Echo(index);
3891 }
3892
3893 %/mathpiper
3894
3895 %output,preserve="false"
3896 Result: True
3897
3898 Side Effects:
3899 1
3900 2
3901 3
3902 4
3903 6
3904 7
3905 8
3906 . %/output
```

3905 Notice that the number 5 is not printed when this program is executed.

### 3906 19.1.6 The Repeat() Looping Procedure

3907 The **Repeat()** procedure is a looping procedure that is similar to While() and  
3908 ForEach(), but it is simpler than these two. Here are the two calling formats for  
3909 Repeat():

```
Repeat(count) body
Repeat() body
```

3910 The first version of Repeat() simply takes an integer argument that indicates how  
3911 many times it should loop. The following program shows how to use Repeat() to  
3912 print 4 copies of the word "Hello":

```
3913 %mathpiper
3914
3915 Repeat(4)
3916 {
3917     Echo("Hello");
3918 }
3919
3920 %/mathpiper
3921
3922     %output,preserve="false"
3923     Result: 4
3924
3925     Side Effects:
3926     Hello
3927     Hello
3928     Hello
3929     Hello
3930 .    %/output
```

3930 The second version of Repeat() does not take any arguments and it is designed to  
3931 run as an **infinite loop**. The Break() procedure is then used to make the  
3932 Repeat() procedure stop looping. The following program would print the loop  
3933 indexing variable **index** forever, but the Break() procedure is used to stop the  
3934 loop after **3** iterations:

```
3935 %mathpiper
3936
3937 index := 1;
3938
3939 loopCount := Repeat()
3940 {
3941     Echo(index);
3942
3943     If(index == 3) Break();
3944
3945     index := (index + 1);
3946 }
3947
```

```

3948 Echo("Loop count: ", loopCount);
3949
3950 %/mathpiper
3951     %output,preserve="false"
3952     Result: True
3953
3954     Side Effects:
3955     1
3956     2
3957     3
3958     Loop count: 2
3959 . %/output

```

3960 Notice that Repeat() returns the number of times it actually looped as a result  
 3961 and that this value is assigned to the variable **loopCount**.

### 3962 19.1.7 The EchoTime() Procedure

3963 Computers are extremely fast, but they still take time to execute programs.  
 3964 Sometimes it is important to determine how long it takes to evaluate a given  
 3965 expression in order to do things like determine if a section of code need to run  
 3966 quicker than it currently is or determine if one piece of code is slower than  
 3967 another. The EchoTime() procedure is a bodied procedure that is used to **time**  
 3968 **how long a section of code takes to run**. Here is its calling format:

```
EchoTime()expression
```

3969 The following examples use EchoTime() to determine how long it takes to add the  
 3970 numbers 2 and 3 together and how long it takes to factor 1234567:

```

3971 In> EchoTime() 2 + 3
3972 Result: 5
3973 Side Effects:
3974 0.000080946 seconds taken.

3975 In> EchoTime() Factor(1234567)
3976 Result: 127*9721
3977 Side Effects:
3978 0.395028773 seconds taken.

```

3979 In the following program, a ForEach loop is used to have the Factor() procedure  
 3980 factor all the numbers in a list. The EchoTime() procedure is used to determine  
 3981 how long it takes to do all the factoring:

```
3982 %mathpiper
3983
3984 EchoTime() ForEach(number, [100, 54, 65, 67, 344, 98, 454])
3985 {
3986     Echo(number, " - ", Factor(number));
3987 }
3988
3989 %/mathpiper
3990
3990 %output,preserve="false"
3991 Result: True
3992
3993 Side Effects:
3994 100 - 2^2*5^2
3995 54 - 2*3^3
3996 65 - 5*13
3997 67 - 67
3998 344 - 2^3*43
3999 98 - 2*7^2
4000 454 - 2*227
4001 0.262678978 seconds taken.
4002 . %/output
```

4003 Finally, the following program shows how to time a code sequence that prints the  
4004 numbers from 1 to 100:

```
4005 %mathpiper
4006
4007 EchoTime()
4008 {
4009     index := 1;
4010
4011     While(index <=? 100)
4012     {
4013         Write(index, ', ');
4014
4015         If(index % 10 ==? 0) NewLine();
4016
4017         index++;
4018     }
4019
4020     NewLine();
4021 }
4022
4022 %/mathpiper
4023
4023 %output,preserve="false"
4024 Result: True
4025
```

```
4026     Side Effects:
4027     1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
4028     11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
4029     21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
4030     31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
4031     41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
4032     51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
4033     61, 62, 63, 64, 65, 66, 67, 68, 69, 70,
4034     71, 72, 73, 74, 75, 76, 77, 78, 79, 80,
4035     81, 82, 83, 84, 85, 86, 87, 88, 89, 90,
4036     91, 92, 93, 94, 95, 96, 97, 98, 99, 100,
4037
4038     0.055418423 seconds taken.
4039     .    %/output
```

## 4040 19.2 Exercises

4041 For the following exercises, create a new MathPiperIDE worksheet file called  
4042 **book\_1\_section\_19\_exercises\_<your first name>\_<your last name>.mpws.**  
4043 (**Note: there are no spaces in this file name**). For example, John Smith's  
4044 worksheet would be called:

4045 **book\_1\_section\_19\_exercises\_john\_smith.mpws.**

4046 After this worksheet has been created, place your answer for each exercise that  
4047 requires a fold into its own fold in this worksheet. Place a title attribute in the  
4048 start tag of each fold that indicates the exercise the fold contains the solution to.  
4049 The folds you create should look similar to this one:

```
4050 %mathpiper,title="Exercise 1"
```

```
4051 //Sample fold.
```

```
4052 %/mathpiper
```

4053 If an exercise uses the MathPiper console instead of a fold, copy the work you  
4054 did in the console into a text file so it can be saved.

### 4055 19.2.1 Exercise 1

4056 Create a program that uses a While loop to display the numbers from 1 to  
4057 50. Use the ++ operator to increment the loop index variable.

### 4058 19.2.2 Exercise 2

4059 Create a program that uses a While loop to display the numbers from 1 to 50  
4060 in reverse order. Use the -- operator to decrement the loop index  
4061 variable.

**4062 19.2.3 Exercise 3**

4063 Create a program that uses a Continue() procedure to cause a While loop  
4064 that is configured to print the numbers from 1 to 100 to skip printing the  
4065 number 72.

**4066 19.2.4 Exercise 4**

4067 Create a program that uses the version of the Repeat() procedure that takes  
4068 an integer as an argument and the + string concatenation operator to print  
4069 the following:

4070 Hello  
4071 HelloHello  
4072 HelloHelloHello  
4073 HelloHelloHelloHello  
4074 HelloHelloHelloHelloHello

4075 Hint:

4076 In> string := "Hi"  
4077 Result: "Hi"

4078 In> string := string + "Hi"  
4079 Result: "HiHi"

**4080 19.2.5 Exercise 5**

4081 In the last example in the EchoTime() section, what operator is being used  
4082 to format the output into lines of 10 numbers and how is this operator  
4083 doing this?