

MathRider For Newbies

by Ted Kosan

Copyright © 2008 by Ted Kosan

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

Table of Contents

1	Preface.....	9
1.1	Dedication.....	9
1.2	Acknowledgments.....	9
1.3	Support Email List.....	9
2	Introduction.....	10
2.1	What Is A Super Scientific Calculator?.....	10
2.2	What Is MathRider?.....	11
2.3	What Inspired The Creation Of Mathrider?.....	12
3	Downloading And Installing MathRider.....	14
3.1	Installing Sun's Java Implementation.....	14
3.1.1	Installing Java On A Windows PC.....	14
3.1.2	Installing Java On A Macintosh.....	14
3.1.3	Installing Java On A Linux PC.....	14
3.2	Downloading And Extracting.....	15
3.2.1	Extracting The Archive File For Windows Users.....	16
3.2.2	Extracting The Archive File For Unix Users.....	16
3.3	MathRider's Directory Structure & Execution Instructions.....	16
3.3.1	Executing MathRider On Windows Systems.....	17
3.3.2	Executing MathRider On Unix Systems.....	17
3.3.2.1	MacOS X.....	17
4	The Graphical User Interface.....	18
4.1	Buffers And Text Areas.....	18
4.2	The Gutter.....	18
4.3	Menus.....	18
4.3.1	File.....	18
4.3.2	Edit.....	19
4.3.3	Search.....	19
4.3.4	Markers.....	19
4.3.5	Folding.....	20
4.3.6	View.....	20
4.3.7	Utilities.....	20
4.3.8	Macros.....	21
4.3.9	Plugins.....	21
4.3.10	Help.....	21
4.4	The Toolbar.....	21
5	MathRider's Plugin-Based Extension Mechanism.....	22
5.1	What Is A Plugin?.....	22

5.2 Which Plugins Are Currently Included When MathRider Is Installed?.....	22
5.3 What Kinds Of Plugins Are Possible?.....	23
5.3.1 Plugins Based On Java Applets.....	23
5.3.2 Plugins Based On Java Applications.....	23
5.3.3 Plugins Which Talk To Native Applications.....	23
6 Exploring The MathRider Application.....	24
6.1 The Console.....	24
6.2 MathPiper Program Files.....	24
6.3 MathRider Worksheets.....	24
6.4 Plugins.....	24
7 MathPiper: A Computer Algebra System For Beginners.....	26
7.1 Numeric Vs. Symbolic Computations.....	26
7.1.1 Using The MathPiper Console As A Numeric (Scientific) Calculator..	27
7.1.1.1 Functions.....	28
7.1.1.2 Accessing Previous Input And Results.....	29
7.1.1.3 Syntax Errors.....	29
7.1.2 Using The MathPiper Console As A Symbolic Calculator.....	30
7.1.2.1 Variables.....	30
8 The MathPiper Documentation Plugin.....	34
8.1 Function List.....	34
8.2 Mini Web Browser Interface.....	34
9 Using MathRider As A Programmer's Text Editor.....	36
9.1 Creating, Opening, And Saving Text Files.....	36
9.2 Editing Files.....	36
9.2.1 Rectangular Selection Mode.....	36
9.3 File Modes.....	37
9.4 Entering And Executing Stand Alone MathPiper Programs.....	37
10 MathRider Worksheet Files.....	38
10.1 Code Folds.....	38
10.2 Fold Properties.....	39
10.3 Currently Implemented Fold Types And Properties.....	40
10.3.1 %geogebra & %geogebra_xml.....	40
10.3.2 %hoteqn.....	43
10.3.3 %mathpiper.....	44
10.3.3.1 Plotting MathPiper Functions With GeoGebra.....	45
10.3.3.2 Displaying MathPiper Expressions In Traditional Form With HotEqn.....	46
10.3.4 %output.....	46
10.3.5 %error.....	46
10.3.6 %html.....	46

10.3.7 %beanshell.....	49
10.4 Automatically Inserting Folds & Removing Unpreserved Folds.....	49
11 MathPiper Programming Fundamentals.....	50
11.1 Values and Expressions.....	50
11.2 Operators.....	50
11.3 Operator Precedence.....	51
11.4 Changing The Order Of Operations In An Expression.....	52
11.5 Variables.....	53
11.6 Functions & Function Names.....	54
11.7 Functions That Produce Side Effects.....	55
11.7.1 The Echo() and Write() Functions.....	55
11.7.1.1 Echo().....	55
11.7.1.2 Write().....	57
11.8 Expressions Are Separated By Semicolons.....	58
11.9 Strings.....	59
11.10 Comments.....	60
11.11 Conditional Operators.....	61
11.12 Making Decisions With The If() Function & Predicate Expressions.....	63
11.13 The And(), Or(), & Not() Boolean Functions & Infix Notation.....	65
11.13.1 And().....	65
11.13.2 Or().....	67
11.13.3 Not() & Prefix Notation.....	68
11.14 The While() Looping Function & Bodied Notation.....	69
11.15 Long-Running Loops, Infinite Loops, & Interrupting Execution.....	73
11.16 Predicate Functions.....	74
11.17 Lists: Values That Hold Sequences Of Expressions.....	75
11.17.1 Using While() Loops With Lists	77
11.17.2 The ForEach() Looping Function.....	78
11.18 Functions & Operators Which Loop Internally To Process Lists.....	79
11.18.1 TableForm().....	79
11.18.2 The .. Range Operator.....	79
11.18.3 Contains().....	80
11.18.4 Find().....	80
11.18.5 Count().....	81
11.18.6 Select().....	81
11.18.7 The Nth() Function & The [] Operator.....	82
11.18.8 Append() & Nondestructive List Operations.....	82
11.18.9 The : Prepend Operator.....	83
11.18.10 Concat().....	84
11.18.11 Insert(), Delete(), & Replace().....	84
11.18.12 Take()	84
11.18.13 Drop().....	85

11.18.14 FillList().....	86
11.18.15 RemoveDuplicates().....	86
11.18.16 Reverse().....	87
11.18.17 Partition().....	87
11.19 Functions That Work With Integers.....	88
11.19.1 RandomIntegerVector().....	88
11.19.2 Max() & Min().....	88
11.19.3 Div() & Mod().....	89
11.19.4 Gcd().....	89
11.19.5 Lcm().....	90
11.19.6 Add().....	90
11.19.7 Factorize().....	91
11.20 User Defined Functions.....	91
11.20.1 Global Variables, Local Variables, & Local().....	93
11.21 Applying Functions To List Members.....	95
11.21.1 Table()	95
12 THE CONTENT BELOW THIS LINE IS STILL UNDER DEVELOPMENT.....	96
12.1 Sets.....	96
13 Miscellaneous Topics.....	97
13.1 Errors.....	97
13.2 Style Guide For Expressions.....	97
13.3 Built-in Constants.....	97
14 Solving Equations.....	98
14.1 Solving Equations Symbolically.....	98
14.1.1 Symbolic Expressions & Simplify().....	98
14.1.1.1 Expanding And Factoring.....	98
14.1.1.2 Miscellaneous Symbolic Expression Examples.....	99
14.1.2 Symbolic Equations and The solve() Function.....	99
14.2 Solving Equations Numerically.....	100
14.2.1 Roots.....	100
14.3 Finding Roots Graphically And Numerically With The find_root() Method	101
15 Output Forms.....	102
15.1 LaTeX Is Used To Display Objects In Traditional Mathematics Form....	102
15.2 Displaying Mathematical Objects In Traditional Form.....	102
16 2D Plotting.....	103
17 High School Math Problems (most of the problems are still in development)	104
17.1 Pre-Algebra.....	104
17.1.1 Equations.....	104

17.1.2 Expressions.....	104
17.1.3 Geometry.....	104
17.1.4 Inequalities.....	104
17.1.5 Linear Functions.....	104
17.1.6 Measurement.....	104
17.1.7 Nonlinear Functions.....	105
17.1.8 Number Sense And Operations.....	105
17.1.8.1 Express an integer fraction in lowest terms.....	105
17.1.9 Polynomial Functions.....	106
17.2 Algebra.....	106
17.2.1 Absolute Value Functions.....	106
17.2.2 Complex Numbers.....	106
17.2.3 Composite Functions.....	106
17.2.4 Conics.....	106
17.2.5 Data Analysis.....	106
17.2.6 Discrete Mathematics	107
17.2.7 Equations.....	107
17.2.7.1 Express a symbolic fraction in lowest terms.....	107
17.2.7.2 Determine the product of two symbolic fractions.....	108
17.2.7.3 Solve a linear equation for x.....	109
17.2.7.4 Solve a linear equation which has fractions.....	110
17.2.8 Exponential Functions.....	112
17.2.9 Exponents.....	112
17.2.10 Expressions.....	112
17.2.11 Inequalities.....	112
17.2.12 Inverse Functions.....	112
17.2.13 Linear Equations And Functions.....	112
17.2.14 Linear Programming.....	112
17.2.15 Logarithmic Functions.....	113
17.2.16 Logistic Functions.....	113
17.2.17 Matrices.....	113
17.2.18 Parametric Equations.....	113
17.2.19 Piecewise Functions.....	113
17.2.20 Polynomial Functions.....	113
17.2.21 Power Functions.....	113
17.2.22 Quadratic Functions.....	113
17.2.23 Radical Functions.....	114
17.2.24 Rational Functions.....	114
17.2.25 Sequences.....	114
17.2.26 Series.....	114
17.2.27 Systems of Equations.....	114
17.2.28 Transformations.....	114

17.2.29 Trigonometric Functions.....	114
17.3 Precalculus And Trigonometry.....	114
17.3.1 Binomial Theorem.....	115
17.3.2 Complex Numbers.....	115
17.3.3 Composite Functions.....	115
17.3.4 Conics.....	115
17.3.5 Data Analysis.....	115
17.3.6 Discrete Mathematics.....	115
17.3.7 Equations.....	115
17.3.8 Exponential Functions.....	115
17.3.9 Inverse Functions.....	116
17.3.10 Logarithmic Functions.....	116
17.3.11 Logistic Functions.....	116
17.3.12 Mathematical Analysis.....	116
17.3.13 Matrices And Matrix Algebra.....	116
17.3.14 Parametric Equations.....	116
17.3.15 Piecewise Functions.....	116
17.3.16 Polar Equations.....	116
17.3.17 Polynomial Functions.....	117
17.3.18 Power Functions.....	117
17.3.19 Quadratic Functions.....	117
17.3.20 Radical Functions.....	117
17.3.21 Rational Functions.....	117
17.3.22 Real Numbers.....	117
17.3.23 Sequences.....	117
17.3.24 Series.....	117
17.3.25 Sets.....	118
17.3.26 Systems of Equations.....	118
17.3.27 Transformations.....	118
17.3.28 Trigonometric Functions.....	118
17.3.29 Vectors.....	118
17.4 Calculus.....	118
17.4.1 Derivatives.....	118
17.4.2 Integrals.....	118
17.4.3 Limits.....	119
17.4.4 Polynomial Approximations And Series.....	119
17.5 Statistics.....	119
17.5.1 Data Analysis.....	119
17.5.2 Inferential Statistics.....	119
17.5.3 Normal Distributions.....	119
17.5.4 One Variable Analysis.....	119
17.5.5 Probability And Simulation.....	119

17.5.6 Two Variable Analysis.....	120
18 High School Science Problems.....	121
18.1 Physics.....	121
18.1.1 Atomic Physics.....	121
18.1.2 Boiling.....	121
18.1.3 Buoyancy.....	121
18.1.4 Circular Motion.....	121
18.1.5 Convection.....	121
18.1.6 Density.....	121
18.1.7 Diffusion.....	122
18.1.8 Dynamics.....	122
18.1.9 Electricity And Magnetism.....	122
18.1.10 Energy.....	122
18.1.11 Fluids.....	122
18.1.12 Freezing.....	122
18.1.13 Friction.....	122
18.1.14 Heat Transfer.....	122
18.1.15 Insulation.....	123
18.1.16 Kinematics.....	123
18.1.17 Light.....	123
18.1.18 Momentum.....	123
18.1.19 Newton's Laws.....	123
18.1.20 Optics.....	123
18.1.21 Pressure.....	123
18.1.22 Pulleys.....	123
18.1.23 Relativity.....	124
18.1.24 Rotational Motion.....	124
18.1.25 Sound.....	124
18.1.26 Thermodynamics.....	124
18.1.27 Waves.....	124
18.1.28 Work.....	124

1 Preface

2 1.1 Dedication

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

5 1.2 Acknowledgments

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

8 Susan Addington

9 Matthew Moelter

10 Sherm Ostrowsky

11 1.3 Support Email List

12 The support email list for this book is called **mathrider-**
13 **users@googlegroups.com** and you can subscribe to it at
14 <http://groups.google.com/group/mathrider-users>. Please place **[Newbies book]**
15 in the title of your email when you post to this list if the topic of the post is
16 related to this book.

17 2 Introduction

18 MathRider is an open source Super Scientific Calculator (SSC) for performing
19 [numeric and symbolic computations](#). Super scientific calculators are complex
20 and it takes a significant amount of time and effort to become proficient at using
21 one. The amount of power that a super scientific calculator makes available to a
22 user, however, is well worth the effort needed to learn one. It will take a
23 beginner a while to become an expert at using MathRider, but fortunately one
24 does not need to be a MathRider expert in order to begin using it to solve
25 problems.

26 2.1 What Is A Super Scientific Calculator?

27 A super scientific calculator is a set of computer programs that 1) automatically
28 perform a wide range of numeric and symbolic mathematics calculation
29 algorithms and 2) provide a user interface which enables the user to access
30 these calculation algorithms and manipulate the mathematical object they
31 create.

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

38 Calculation algorithms exist for many areas of mathematics and new algorithms
39 are constantly being developed. Another name for this kind of software is a
40 Computer Algebra System (CAS). A significant number of computer algebra
41 systems have been created since the 1960s and the following list contains some
42 of the more popular ones:

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

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

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

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

51 and here is the same formula in text form:

52 $a = x^2 + 4*h*x + 3/7$

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

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

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

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

74 **2.2 What Is MathRider?**

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

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

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

Table 1: MathRider user experience levels.

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

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

92 **2.3 What Inspired The Creation Of Mathrider?**

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

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

96 and Steve Yegge's thoughts on learning mathematics:

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

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

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

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

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

109 **3 Downloading And Installing MathRider**

110 **3.1 *Installing Sun's Java Implementation***

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

116 **3.1.1 Installing Java On A Windows PC**

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

120 <http://java.com/>

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

123 **3.1.2 Installing Java On A Macintosh**

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

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

128 **3.1.3 Installing Java On A Linux PC**

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

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

140 You should also be aware that a number of Linux distributions distribute a non-
141 Sun implementation of Java which is not 100% compatible with it. Running
142 sophisticated GUI-based Java programs on a non-Sun version of Java usually does

143 not work. In order to check to see what version of Java you have installed (if
144 any), execute the following command in a shell (MathRider needs at least Java
145 5):

146 java -version

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

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

153 **3.2 Downloading And Extracting**

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

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

161 <http://mathrider.org>

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

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

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

178 The archive file can be easily copied to a CD or USB drive if you would like to
179 install MathRider on another computer or give it to a friend.

180 **(Note: If you already have a version of MathRider installed and you want**

181 **to install a new version in the same directory that holds the old version,**
182 **you must delete the old version first or move it to a separate directory.)**

183 3.2.1 Extracting The Archive File For Windows Users

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

188 After the extraction process is complete, a new folder called **mathrider** should
189 be present in the same folder that contains the archive file.

190 3.2.2 Extracting The Archive File For Unix Users

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

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

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

197 3.3 MathRider's Directory Structure & Execution Instructions

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

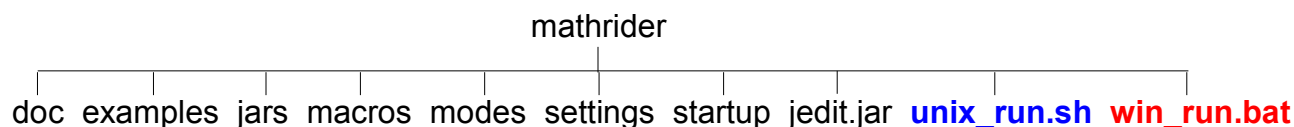


Illustration 1: MathRider's Directory Structure

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

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

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

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

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

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

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

208 **startup** - Contains startup scripts that are executed each time MathRider
209 launches.

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

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

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

213 3.3.1 Executing MathRider On Windows Systems

214 Open the **mathrider** folder and double click on the **win_run** file.

215 3.3.2 Executing MathRider On Unix Systems

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

218 sh unix_run.sh

219 3.3.2.1 MacOS X

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

223 cd /Applications/mathrider

224 Run mathrider by typing:

225 sh unix_run.sh

226 4 The Graphical User Interface

227 MathRider is built on top of jEdit (<http://jedit.org>) so it has the "heart" of a
228 programmer's text editor. Text editors are similar to standard text editors and
229 word processors in a number of ways so getting started with MathRider should
230 be relatively easy for anyone who has used either one of these. Don't be fooled,
231 though, because programmer's text editors have capabilities that are far more
232 advanced than any standard text editor or word processor.

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

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

241 4.1 Buffers And Text Areas

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

248 4.2 The Gutter

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

252 4.3 Menus

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

259 4.3.1 File

260 The File menu contains actions which are typically found in normal text editors

261 and word processors. The actions to create new files, save files, and open
262 existing files are all present along with variations on these actions.
263 Actions for opening recent files, configuring the page setup, and printing are
264 also present.

265 4.3.2 Edit

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

271 4.3.3 Search

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

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

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

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

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

296 4.3.4 Markers

297 The Markers menu contains actions which place markers into a buffer, removes

298 them, and scrolls the document to them when they are selected. When a marker
299 is placed into a buffer, a link to it will be added to the bottom of the Markers
300 menu. Selecting a marker link will scroll the buffer to the marker it points to.
301 The list of marker links are kept in a temporary file which is placed into the same
302 directory as the buffer's file.

303 4.3.5 Folding

304 A **fold** is a section of a buffer that can be hidden (folded) or shown (unfolded) as
305 needed. In [worksheet files](#) (which have a .mrw extension) folds are created by
306 wrapping sections of a buffer in tags. For example, HTML folds start with a
307 %html tag and end with an %/html tag. See the **worksheet_demo_1.mws** file
308 for examples of folds.

309 Folds are folded and unfolded by pressing on the small black triangles that are
310 next to each fold in the [gutter](#).

311 4.3.6 View

312 A **view** is a copy of the complete MathRider application window. It is possible to
313 create multiple views if numerous buffers are being edited, multiple plugins are
314 being used, etc. The top part of the **View** menu contains actions which allow
315 views to be opened and closed but most beginners will only need to use a single
316 view.

317 The middle part of the **View** menu allows the user to navigate between buffers,
318 and the bottom part of the menu contains a **Scrolling** sub-menu, a **Splitting**
319 sub-menu, and a **Docking** sub-menu.

320 The **Scrolling** sub-menu contains actions for scrolling a text area.

321 The **Splitting** sub-menu contains actions which allow a text area to be split into
322 multiple sections so that different parts of a buffer can be edited at the same
323 time. When you are done using a split view of a buffer, select the **Unsplit All**
324 action and the buffer will be shown in a single text area again.

325 The **Docking** sub-menu allows plugins to be attached to the top, bottom, left,
326 and right sides of the main window. Plugins can even be made to float free of the
327 main window in their own separate window. Plugins and their docking
328 capabilities are covered in the [Plugins](#) section of this document.

329 4.3.7 Utilities

330 The utilities menu contains a significant number of actions, some that are useful
331 to beginners and others that are meant for experts. The two actions that are
332 most useful to beginners are the **Buffer Options** actions and the **Global**
333 **Options** actions. The **Buffer Options** actions allows the currently selected
334 buffer to be customized and the **Global Options** actions brings up a rich dialog

335 window that allows numerous aspects of the MathRider application to be
336 configured.

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

338 **4.3.8 Macros**

339 **Macros** are small programs that perform useful tasks for the user. The top of
340 the **Macros** menu contains actions which allow macros to be created by
341 recording a sequence of user steps which can be saved for later execution. The
342 bottom of the **Macros** menu contains macros that can be executed as needed.

343 The main language that MathRider uses for macros is called **BeanShell** and it is
344 based upon Java's syntax. Significant parts of MathRider are written in
345 BeanShell, including many of the actions which are present in the menus. After
346 a user knows how to program in BeanShell, it can be used to easily customize
347 (and even extend) MathRider.

348 **4.3.9 Plugins**

349 Plugins are component-like pieces of software that are designed to provide an
350 application with extended capabilities and they are similar in concept to physical
351 world components. See the [plugins](#) section for more information about plugins.

352 **4.3.10 Help**

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

356 **4.4 The Toolbar**

357 The **Toolbar** is located just beneath the menus near the top of the main window
358 and it contains a number of icon-based buttons. These buttons allow the user to
359 access the same actions which are accessible through the menus just by clicking
360 on them. There is not room on the toolbar for all the actions in the menus to be
361 displayed, but the most common actions are present. The user also has the
362 option of customizing the toolbar by using the **Utilities->Global Options->Tool**
363 **Bar** dialog.

364 5 MathRider's Plugin-Based Extension Mechanism

365 5.1 What Is A Plugin?

366 As indicated in a previous section, plugins are component-like pieces of software
367 that are designed to provide an application with extended capabilities and they
368 are similar in concept to physical world components. As an example, think of a
369 plain automobile that is about to have improvements added to it. The owner
370 might plug in a stereo system, speakers, a larger engine, anti-sway bars, wider
371 tires, etc. MathRider can be improved in a similar manner by allowing the user
372 to select plugins from the Internet which will then be downloaded and installed
373 automatically.

374 Most of MathRider's significant power and flexibility are derived from its plugin-
375 based extension mechanism (which it inherits from its jEdit "heart").

376 5.2 Which Plugins Are Currently Included When MathRider Is Installed?

377 **Code2HTML** - Converts a text area into HTML format (complete with syntax
378 highlighting) so it can be published on the web.

379 **Console** - Contains **shell** or **command line** interfaces to various pieces of
380 software. There is a shell for talking with the operating system, one for talking
381 to BeanShell, and one for talking with MathPiper. Additional shells can be added
382 to the Console as needed.

383 **Calculator** - An RPN (Reverse Polish Notation) calculator.

384 **ErrorList** - Provides a short description of errors which were encountered in
385 executed code along with the line number that each error is on. Clicking on an
386 error highlights the line the error occurred on in a text area.

387 **GeoGebra** - Interactive geometry software. MathRider also uses it as an
388 interactive plotting package.

389 **HotEqn** - Renders [LaTeX](#) code.

390 **MathPiper** - A computer algebra system that is suitable for beginners.

391 **LaTeX Tools** - Tools to help automate LaTeX editing tasks.

392 **Project Viewer** - Allows groups of files to be defined as projects.

393 **QuickNotepad** - A persistent text area which notes can be entered into.

394 **SideKick** - Used by plugins to display various buffer structures. For example, a
395 buffer may contain a language which has a number of function definitions and
396 the SideKick plugin would be able to show the function names in a tree.

397 **MathPiperDocs** - Documentation for MathPiper which can be navigated using a
398 simple browser interface.

399 **5.3 What Kinds Of Plugins Are Possible?**

400 Almost any application that can run on the Java platform can be made into a
401 plugin. However, most plugins should fall into one of the following categories:

402 **5.3.1 Plugins Based On Java Applets**

403 Java applets are programs that run inside of a web browser. Thousands of
404 mathematics, science, and technology-oriented applets have been written since
405 the mid 1990s and most of these applets can be made into a MathRider plugin.

406 **5.3.2 Plugins Based On Java Applications**

407 Almost any Java-based application can be made into a MathRider plugin.

408 **5.3.3 Plugins Which Talk To Native Applications**

409 A native application is one that is not written in Java and which runs on the
410 computer being used. Plugins can be written which will allow MathRider to
411 interact with most native applications.

412 6 Exploring The MathRider Application

413 6.1 The Console

414 The lower left window contains consoles. Switch to the MathPiper console by
415 pressing the small black inverted triangle which is near the word **System**.
416 Select the MathPiper console and when it comes up, enter simple **mathematical**
417 **expressions** (such as 2+2 and 3*7) and execute them by pressing **<enter>**
418 (**expressions** are explained in section [11. MathPiper Programming](#)
419 [Fundamentals](#)).

420 6.2 MathPiper Program Files

421 The MathPiper programs in the text window (which have **.mpi** extensions) can
422 be executed by placing the cursor in a window and pressing **<shift><enter>**.
423 The output will be displayed in the MathPiper console window.

424 6.3 MathRider Worksheets

425 The most interesting files are MathRider **worksheet** files (which are the ones
426 that end with a **.mrw** extension). MathRider worksheets consist of **folds** which
427 contain different types of code that can be executed by pressing
428 **<shift><enter>** inside of them. Select the **worksheet_demo_1.mrw** tab and
429 follow the instructions which are present within the comments it contains.

430 6.4 Plugins

431 At the right side of the application is a small tab that has **Jung** written on it.
432 Press this tab a number of times to see what happens (Jung should be shown and
433 hidden as you press the tab.)

434 The right side of the application also contains a plugin called MathPiperDocs.
435 Open the plugin and look through the documentation by pressing the hyperlinks.
436 You can go back to the main documentation page by pressing the **Home** icon
437 which is at the top of the plugin. Pressing on a function name in the list box will
438 display the documentation for that function.

439 The tabs at the bottom of the screen which read **Activity Log**, **Console**, and
440 **Error List** are all plugins that can be shown and hidden as needed.

441 Go back to the Jung plugin and press the small black inverted triangle that is
442 near it. A pop up menu will appear which has menu items named **Float**, **Dock at**
443 **Top**, etc. Select the **Float** menu item and see what happens.

444 The Jung plugin was detached from the main window so it can be resized and
445 placed wherever it is needed. Select the inverted black triangle on the floating

446 windows and try docking the Jung plugin back to the main window again,
447 perhaps in a different position.

448 Try moving the plugins at the bottom of the screen around the same way. If you
449 close a floating plugin, it can be opened again by selecting it from the Plugins
450 menu at the top of the application.

451 Go to the "Plugins" menu at the top of the screen and select the Calculator
452 plugin. You can also play with docking and undocking it if you would like.

453 Finally, whatever position the plugins are in when you close MathRider, they will
454 be preserved when it is launched again.

455 **7 MathPiper: A Computer Algebra System For Beginners**

456 Computer algebra system plugins are among the most exciting and powerful
457 plugins that can be used with MathRider. In fact, computer algebra systems are
458 so important that one of the reasons for creating MathRider was to provide a
459 vehicle for delivering a computer algebra system to as many people as possible.
460 If you like using a scientific calculator, you should love using a computer algebra
461 system!

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

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

474 **7.1 Numeric Vs. Symbolic Computations**

475 A Computer Algebra System (CAS) is software which is capable of performing
476 both numeric and symbolic computations. Numeric computations are performed
477 exclusively with numerals and these are the type of computations that are
478 performed by typical hand-held calculators.

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

484 Richard Fateman, who helped develop the Macsyma computer algebra system,
485 describes the difference between numeric and symbolic computation as follows:

486 What makes a symbolic computing system distinct from a non-symbolic (or
487 numeric) one? We can give one general characterization: the questions one
488 asks and the resulting answers one expects, are irregular in some way. That
489 is, their "complexity" may be larger and their sizes may be unpredictable. For
490 example, if one somehow asks a numeric program to "solve for x in the
491 equation $\sin(x) = 0$ " it is plausible that the answer will be some 32-bit
492 quantity that we could print as 0.0. There is generally no way for such a
493 program to give an answer $\{n\pi | integer(n)\}$. A program that could provide
494 this more elaborate symbolic, non-numeric, parametric answer dominates the

merely numerical from a mathematical perspective. The single numerical answer might be a suitable result for some purposes: it is simple, but it is a compromise. If the problem-solving environment requires computing that includes asking and answering questions about sets, functions, expressions (polynomials, algebraic expressions), geometric domains, derivations, theorems, or proofs, then it is plausible that the tools in a symbolic computing system will be of some use.

Problem Solving Environments and Symbolic Computing: Richard J. Fateman:
<http://www.cs.berkeley.edu/~fateman/papers/pse.pdf>

Since most people who read this document will probably be familiar with performing numeric calculations as done on a scientific calculator, the next section shows how to use MathPiper as a scientific calculator. The section after that then shows how to use MathPiper as a symbolic calculator. Both sections use the console interface to MathPiper. In MathRider, a console interface to any plugin or application is a **shell** or **command line** interface to it.

7.1.1 Using The MathPiper Console As A Numeric (Scientific) Calculator

Open the Console plugin by selecting the **Console** tab in the lower left part of the MathRider application. A text area will appear and in the upper left corner of this text area will be a pull down menu which is set to "System". Select this pull down menu and then select the **MathPiper** menu item that is inside of it (feel free to increase the size of the console text area if you would like). When the MathPiper console is first launched, it prints a welcome message and then provides **In>** as an input prompt:

```
MathPiper, a computer algebra system for beginners.
```

```
In>
```

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

```
In> 2+2
Result> 4
```

```
In>
```

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

In addition to addition, MathPiper can also do subtraction, multiplication,

532 exponents, and division:

```
533 In> 5-2  
534 Result> 3
```

```
535 In> 3*4  
536 Result> 12
```

```
537 In> 2^3  
538 Result> 8
```

```
539 In> 12/6  
540 Result> 2
```

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

545 MathPiper can also work with decimal numbers:

```
546 In> .5+1.2  
547 Result> 1.7
```

```
548 In> 3.7-2.6  
549 Result> 1.1
```

```
550 In> 2.2*3.9  
551 Result> 8.58
```

```
552 In> 2.2^3  
553 Result> 10.648
```

```
554 In> 9.5/3.2  
555 Result> 9.5/3.2
```

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

```
559 In> N(9.5/3.2)  
560 Result> 2.96875
```

561 7.1.1.1 Functions

562 **N()** is an example of a **function**. A function can be thought of as a "black box"
563 which accepts input, processes the input, and returns a result. Each function
564 has a name and in this case, the name of the function is **N** which stands for
565 **Numeric**. To the right of a function's name there is always a set of parentheses

566 and information that is sent to the function is placed inside of them. The purpose
567 of the N() function is to make sure that the information that is sent to it is
568 processed numerically instead of symbolically.

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

```
571 In> IsEven(4)
572 Result> True
```

```
573 In> IsEven(5)
574 Result> False
```

575 MathPiper has a large number of functions some of which are described in more
576 depth in the [MathPiper Documentation](#) section and the [MathPiper Programming](#)
577 [Fundamentals](#) section. **A complete list of MathPiper's functions can be**
578 **found in the MathPiperDocs plugin.**

579 **7.1.1.2 Accessing Previous Input And Results**

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

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

```
586 In> 5*8
587 Result> 40
```

```
588 In> %
589 Result> 40
```

```
590 In> %*2
591 Result> 80
```

592 **7.1.1.3 Syntax Errors**

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

```
598 In> 12 \ 6
```

```
599 Error parsing expression, near token \
```

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

603 This section provided a short introduction to using MathPiper as a numeric
604 calculator and the next section contains a short introduction to using MathPiper
605 as a symbolic calculator.

606 7.1.2 Using The MathPiper Console As A Symbolic Calculator

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

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

```
612 In> 1/2 + 1/3
```

```
613 Result> 5/6
```

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

```
618 In> %
```

```
619 Result> 5/6
```

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

```
622 In> Numer(%)
```

```
623 Result> 5
```

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

627 7.1.2.1 Variables

628 What would be nice is if MathPiper provided a way to store **results** (which are
629 also called **values**) in symbols that we choose instead of ones that it chooses.

630 Fortunately, this is exactly what it does! Symbols that can be associated with
631 values are called **variables**. Variable names must start with an upper or lower
632 case letter and be followed by zero or more upper case letters, lower case
633 letters, or numbers. Examples of variable names include: 'a', 'b', 'x', 'y', 'answer',
634 'totalAmount', and 'loop6'.

635 The process of associating a value with a variable is called **assigning** or **binding**
636 the value to the variable. Lets recalculate $\frac{1}{2} + \frac{1}{3}$ but this time we will assign the
637 result to the variable 'a':

```
638 In> a := 1/2 + 1/3
639 Result> 5/6
```

```
640 In> a
641 Result> 5/6
```

```
642 In> Numer(a)
643 Result> 5
```

```
644 In> Denom(a)
645 Result> 6
```

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

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

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

```
654 In> a := 9
655 Result> 9
```

```
656 In> a
657 Result> 9
```

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

```
660 In> Clear(a)
661 Result> True
```

```
662 In> a
663 Result> a
```

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

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

```
673 In> Factor(8)
674 Result> 2^3
```

```
675 In> Factor(14)
676 Result> 2*7
```

```
677 In> Factor(2343)
678 Result> 3*11*71
```

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

```
680 In> x
681 Result> x
```

```
682 In> IsBound(x)
683 Result> False
```

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

```
686 In> Expand(%)
687 Result> x^2+24*x+80
```

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

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

700 Now that it has been shown how to use the MathPiper console as both a

701 **symbolic** and a **numeric** calculator, we are ready to dig deeper into MathPiper.
702 As you will soon discover, MathPiper contains an amazing number of functions
703 which deal with a wide range of mathematics.

704 **8 The MathPiper Documentation Plugin**

705 MathPiper has a significant amount of reference documentation written for it
706 and this documentation has been placed into a plugin called **MathPiperDocs** in
707 order to make it easier to navigate. The left side of the plugin window contains
708 the names of all the functions that come with MathPiper and the right side of the
709 window contains a mini-browser that can be used to navigate the documentation.

710 **8.1 Function List**

711 MathPiper's functions are divided into two main categories called **user** functions
712 and **programmer** functions. In general, the **user functions** are used for
713 solving problems in the MathPiper console or with short programs and the
714 **programmer functions** are used for longer programs. However, users will
715 often use some of the programmer functions and programmers will use the user
716 functions as needed.

717 Both the user and programmer function names have been placed into a tree on
718 the left side of the plugin to allow for easy navigation. The branches of the
719 function tree can be open and closed by clicking on the small "circle with a line
720 attached to it" symbol which is to the left of each branch. Both the user and
721 programmer branches have the functions they contain organized into categories
722 and the **top category in each branch** lists all the functions in the branch in
723 **alphabetical order** for quick access. Clicking on a function will bring up
724 documentation about it in the browser window and selecting the **Collapse**
725 button at the top of the plugin will collapse the tree.

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

734 **8.2 Mini Web Browser Interface**

735 MathPiper's reference documentation is in HTML (or web page) format and so
736 the right side of the plugin contains a mini web browser that can be used to
737 navigate through these pages. The browser's home page contains links to the
738 main parts of the MathPiper documentation. As links are selected, the **Back** and
739 **Forward** buttons in the upper right corner of the plugin allow the user to move
740 backward and forward through previously visited pages and the **Home** button
741 navigates back to the home page.

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

745 **9 Using MathRider As A Programmer's Text Editor**

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

751 Computer programmers are among the most intelligent, intense, and creative
752 people in the world and most of their work is done using a programmer's text
753 editor (or something similar to it). One can imagine that the main tool used by
754 this group of people would be a super-tool with all kinds of capabilities that most
755 people would not even suspect.

756 This book only covers a small part of the editing capabilities that MathRider has,
757 but what is covered will allow the user to begin writing programs.

758 **9.1 Creating, Opening, And Saving Text Files**

759 A good way to begin learning how to use MathRider's text editing capabilities is
760 by creating, opening, and saving text files. A text file can be created either by
761 selecting **File->New** from the menu bar or by selecting the icon for this
762 operation on the tool bar. When a new file is created, an empty text area is
763 created for it along with a new tab named **Untitled**. Feel free to create a new
764 text file and type some text into it (even something like alkjdf alksdj fasldj will
765 work).

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

771 **9.2 Editing Files**

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

777 **9.2.1 Rectangular Selection Mode**

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

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

9.3 File Modes

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

MathRider uses a file's extension type to set its text area into a customized **mode** which highlights various parts of its contents. For example, MathPiper programs have a **.pi** extension and the MathPiper demo programs that are pre-loaded in MathRider when it is first downloaded and launched show how the MathPiper mode highlights parts of these programs.

9.4 Entering And Executing Stand Alone MathPiper Programs

A stand alone MathPiper program is simply a text file that has a **.mpi** extension. MathRider comes with some preloaded example MathPiper programs and new MathPiper programs can be created by making a new text file and giving it a **.mpi** extension.

MathPiper programs are executed by placing the cursor in the program's text area and then pressing **<shift><Enter>**. Output from the program is displayed in the MathPiper console but, unlike the MathPiper console (which automatically displays the result of the last evaluation), programs need to use the **Write()** and **Echo()** functions to display output.

Write() is a low level output function which evaluates its input and then displays it unmodified. **Echo()** is a high level output function which evaluates its input, enhances it, and then displays it. These two functions will be covered in the MathPiper programming section.

MathPiper programs and the MathPiper console are designed to work together. Variables which are created in the console are available to a program and variables which are created in a program are available in the console. This allows a user to move back and forth between a program and the console when solving problems.

820 10 MathRider Worksheet Files

821 While MathRider's ability to execute code with consoles and programs provide a
822 significant amount of power to the user, most of MathRider's power is derived
823 from **worksheets**. MathRider worksheets are text files which have a **.mrw**
824 extension and are able to execute multiple types of code in a single text area.
825 The **worksheet_demo_1.mrw** file (which is preloaded in the MathRider
826 environment when it is first launched) demonstrates how a worksheet is able to
827 execute multiple types of code in what are called **code folds**.

828 10.1 Code Folds

829 Code folds are named sections inside a MathRider worksheet which contain
830 source code that can be executed by placing the cursor inside of a given section
831 and pressing **<shift><Enter>**. A fold always starts with **%** followed by the
832 name of the fold type and its end is marked by the text **%/<foldtype>**. For
833 example, here is a MathPiper fold which will print **Hello World!** to the
834 MathPiper console (Note: the line numbers are not part of the program):

```
835 1:%mathpiper
836 2:
837 3:"Hello World!";
838 4:
839 5:%/mathpiper
```

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

```
844 1:%mathpiper
845 2:
846 3:"Hello World!";
847 4:
848 5:%/mathpiper
849 6:
850 7:    %output,preserve="false"
851 8:    Result: "Hello World!"
852 9:    %/output
```

853 The default type of an output fold is **%output** and this one starts at **line 7** and
854 ends on **line 9**. Folds that can be executed have their first and last lines
855 **highlighted** and folds that cannot be executed do not have their first and last
856 lines highlighted. By default, folds of type **%output** have their **preserve**
857 **property** set to **false**. This tells MathRider to overwrite the **%output** fold with a
858 new version during the next execution of its parent.

859 **10.2 Fold Properties**

860 Folds are able to have **properties** passed to them which can be used to associate
 861 additional information with it or to modify its behavior. For example, the **output**
 862 property can be used to set a MathPiper fold's output to what is called **pretty**
 863 form:

```

864 1: %mathpiper, output="pretty"
865 2:
866 3: x^2 + x/2 + 3;
867 4:
868 5: %/mathpiper
869 6:
870 7:   %output, preserve="false"
871 8:   Result: True
872 9:
873 10:   Side effects:
874 11:
875 12:       2    x
876 13:    x  + - + 3
877 14:         2
878 15: %/output

```

879 Pretty form is a way to have text display mathematical expressions that look
 880 similar to the way they would be written on paper. Here is the above expression
 881 in traditional form for comparison:

$$x^2 + \frac{x}{2} + 3$$

882 (Note: MathRider uses MathPiper's **PrettyForm()** function to convert standard
 883 output into pretty form and this function can also be used in the MathPiper
 884 console. The **True** that is displayed in this output comes from the **PrettyForm()**
 885 function.).

886 Properties are placed on the same line as the fold type and they are set equal to
 887 a value by placing an equals sign (=) to the right of the property name followed
 888 by a value inside of quotes. A comma must be placed between the fold name and
 889 the first property and, if more than one property is being set, each one must be
 890 separated by a comma:

```

891 1: %mathpiper, name="example_1", output="pretty"
892 2:
893 3: x^2 + x/2 + 3;
894 4:
895 5: %/mathpiper
896 6:
897 7:   %output, preserve="false"
898 8:   Result: True

```

```

899 9:
900 10:      Side effects:
901 11:
902 12:      2    x
903 13:      x  + - + 3
904 14:          2
905 15:  %/output

```

906 10.3 Currently Implemented Fold Types And Properties

907 This section covers the fold types that are currently implemented in MathRider
 908 along with the properties that can be passed to them.

909 10.3.1 %geogebra & %geogebra_xml.

910 GeoGebra (<http://www.geogebra.org>) is interactive geometry software and
 911 MathRider includes it as a plugin. A **%geogebra** fold sends standard GeoGebra
 912 commands to the GeoGebra plugin and a **%geogebra_xml** fold sends XML-based
 913 commands to it (XML stands for eXtensible Markup Language). The following
 914 example shows a sequence of GeoGebra commands which plot a function and
 915 add a tangent line to it:

```

916 1: %geogebra,clear="true"
917 2:
918 3: //Plot a function.
919 4: f(x)=2*sin(x)
920 5:
921 6: //Add a tangent line to the function.
922 7: a = 2
923 8: (2,0)
924 9: t = Tangent[a, f]
925 10:
926 11: %/geogebra
927 12:
928 13: %output,preserve="false"
929 14:   GeoGebra updated.
930 15: %/output

```

931 If the **clear** property is set to **true**, GeoGebra's drawing pad will be cleared
 932 before the new commands are executed. Illustration 2 shows the GeoGebra
 933 drawing pad after the code in this fold has been executed:

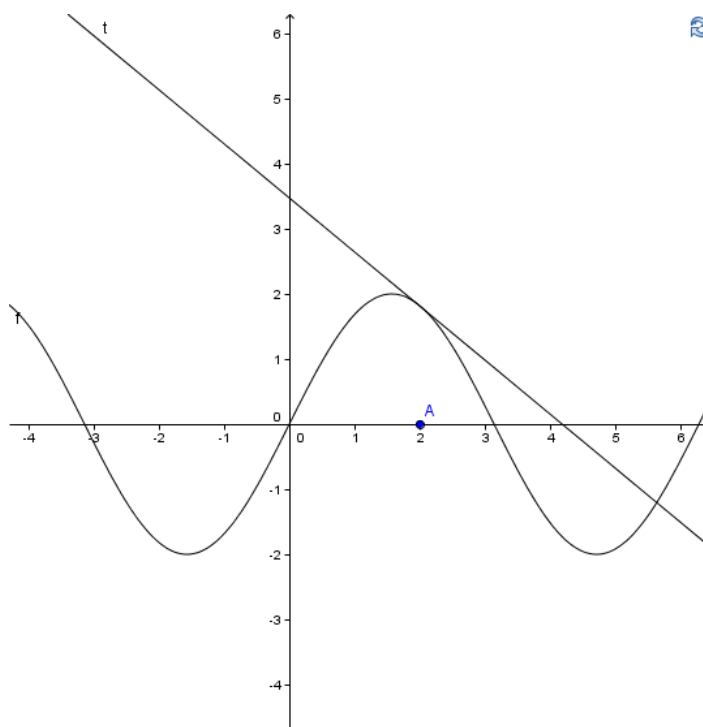


Illustration 2: GeoGebra: $\sin x$ and a tangent to it at $x=2$.

934 GeoGebra saves information in **.ggb** files and these files are compressed **zip** files
 935 which have an **XML** file inside of them. The following XML code was obtained by
 936 adding color information to the previous example, saving it, and unzipping the
 937 .ggb files that was created. The code was then pasted into a **%geogebra_xml**
 938 fold:

```

939 1: %geogebra_xml,description="Obtained from .ggb file"
940 2:
941 3: <?xml version="1.0" encoding="utf-8"?>
942 4: <geogebra format="3.0">
943 5: <gui>
944 6:   <show algebraView="true" auxiliaryObjects="true"
945   algebraInput="true" cmdList="true"/>
946 7:   <splitDivider loc="196" locVertical="400" horizontal="true"/>
947 8:   <font size="12"/>
948 9: </gui>
949 10: <euclidianView>
950 11:   <size width="540" height="553"/>
951 12:   <coordSystem xZero="215.0" yZero="315.0" scale="50.0"
952   yscale="50.0"/>
953 13:   <evSettings axes="true" grid="true" pointCapturing="3"
954   pointStyle="0" rightAngleStyle="1"/>
955 14:   <bgColor r="255" g="255" b="255"/>
956 15:   <axesColor r="0" g="0" b="0"/>

```

```

957 16:    <gridColor r="192" g="192" b="192"/>
958 17:    <lineStyle axes="1" grid="10"/>
959 18:    <axis id="0" show="true" label="" unitLabel="" tickStyle="1"
960    showNumbers="true"/>
961 19:    <axis id="1" show="true" label="" unitLabel="" tickStyle="1"
962    showNumbers="true"/>
963 20:    <grid distX="0.5" distY="0.5"/>
964 21:</euclidianView>
965 22:<kernel>
966 23:    <continuous val="true"/>
967 24:    <decimals val="2"/>
968 25:    <angleUnit val="degree"/>
969 26:    <coordStyle val="0"/>
970 27:</kernel>
971 28:<construction title="" author="" date="">
972 29:<expression label="f" exp="f(x) = 2 sin(x)"/>
973 30:<element type="function" label="f">
974 31:    <show object="true" label="true"/>
975 32:    <objColor r="0" g="0" b="255" alpha="0.0"/>
976 33:    <labelMode val="0"/>
977 34:    <animation step="0.1"/>
978 35:    <fixed val="false"/>
979 36:    <breakpoint val="false"/>
980 37:    <lineStyle thickness="2" type="0"/>
981 38:</element>
982 39:<element type="numeric" label="a">
983 40:    <value val="2.0"/>
984 41:    <show object="false" label="true"/>
985 42:    <objColor r="0" g="0" b="0" alpha="0.1"/>
986 43:    <labelMode val="1"/>
987 44:    <animation step="0.1"/>
988 45:    <fixed val="false"/>
989 46:    <breakpoint val="false"/>
990 47:</element>
991 48:<element type="point" label="A">
992 49:    <show object="true" label="true"/>
993 50:    <objColor r="0" g="0" b="255" alpha="0.0"/>
994 51:    <labelMode val="0"/>
995 52:    <animation step="0.1"/>
996 53:    <fixed val="false"/>
997 54:    <breakpoint val="false"/>
998 55:    <coords x="2.0" y="0.0" z="1.0"/>
999 56:    <coordStyle style="cartesian"/>
1000 57:    <pointSize val="3"/>
1001 58:</element>
1002 59:<command name="Tangent">
1003 60:    <input a0="a" a1="f"/>
1004 61:    <output a0="t"/>
1005 62:</command>
1006 63:<element type="line" label="t">
1007 64:    <show object="true" label="true"/>
1008 65:    <objColor r="255" g="0" b="0" alpha="0.0"/>

```

```

1009 66:      <labelMode val="0"/>
1010 67:      <breakpoint val="false"/>
1011 68:      <coords x="0.8322936730942848" y="1.0" z="-3.4831821998399333"/>
1012 69:      <lineStyle thickness="2" type="0"/>
1013 70:      <eqnStyle style="explicit"/>
1014 71:</element>
1015 72:</construction>
1016 73:</geogebra>
1017 74:
1018 75: %/geogebra_xml
1019 76:
1020 77:      %output,preserve="false"
1021 78:      GeoGebra updated.
1022 79:      %/output

```

1023 Illustration 3 shows the result of sending this XML code to GeoGebra:

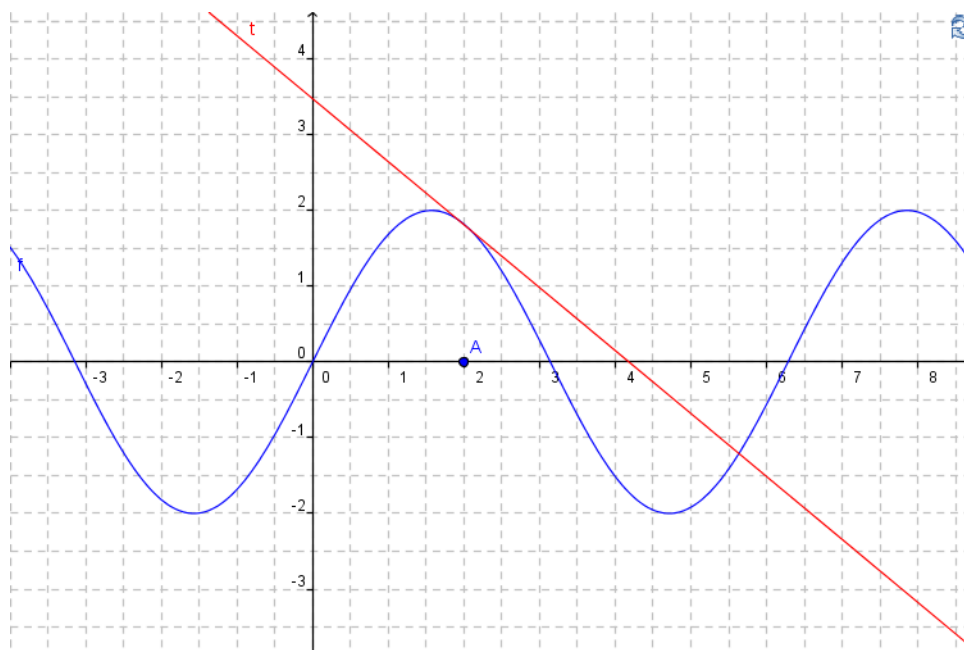


Illustration 3: Generated from %geogebra_xml fold.

1024 **%geogebra_xml** folds are not as easy to work with as plain **%geogebra** folds,
 1025 but they have the advantage of giving the user full control over the GeoGebra
 1026 environment. Both types of folds can be used together while working with
 1027 GeoGebra and this means that the user can send code to the GeoGebra plugin
 1028 from multiple folds during a work session.

1029 10.3.2 %hoteqn

1030 Before understanding what the HotEqn (<http://www.atp.ruhr-uni-bochum.de/VCLab/software/HotEqn/HotEqn.html>) plugin does, one must first
 1031 know a little bit about LaTeX. LaTeX is a **markup language** which allows
 1032

1033 formatting information (such as font size, color, and italics) to be added to plain
 1034 text. LaTeX was designed for creating technical documents and therefore it is
 1035 capable of marking up mathematics-related text. The hoteqn plugin accepts
 1036 input marked up with LaTeX's mathematics-oriented commands and displays it in
 1037 **traditional mathematics** form. For example, to have HotEqn show 2^3 , send it
 1038 $2^{\{3\}}$:

```
1039 1:%hoteqn
1040 2:
1041 3:2^{\{3\}}
1042 4:
1043 5:%/hoteqn
1044 6:
1045 7:    %output,preserve="false"
1046 8:    HotEqn updated.
1047 9:    %/output
```

1048 and it will display:

$$2^3$$

1049 To have HotEqn show $2x^3 + 14x^2 + \frac{24x}{7}$, send it the following code:

```
1050 1:%hoteqn
1051 2:
1052 3:2 x ^{\{3\}} + 14 x ^{\{2\}} + \frac{24 x}{7}
1053 4:
1054 5:%/hoteqn
1055 6:
1056 7:    %output,preserve="false"
1057 8:    HotEqn updated.
1058 9:    %/output
```

1059 and it will display:

$$2x^3 + 14x^2 + \frac{24x}{7}$$

1060 %hoteqn folds are handy for displaying typed-in LaTeX text in traditional form,
 1061 but their main use is to allow other folds to display mathematical objects in
 1062 traditional form. The next section discusses this second use further.

1063 10.3.3 %mathpiper

1064 %mathpiper folds were introduced in a previous section and later sections
 1065 discuss how to start programming in MathPiper. This section shows how

1066 properties can be used to tell %mathpiper folds to generate output that can be
1067 sent to plugins.

1068 **10.3.3.1 Plotting MathPiper Functions With GeoGebra**

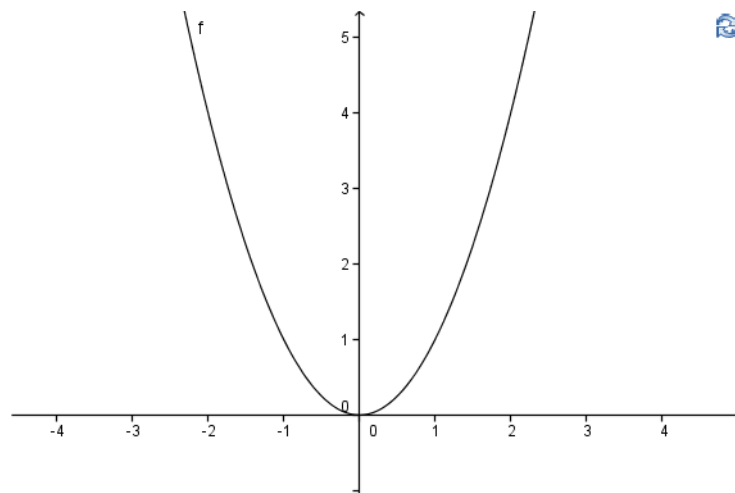
1069 When working with a computer algebra system, a user often needs to plot a
1070 function in order to understand it better. GeoGebra can plot functions and a
1071 %mathpiper fold can be configured to generate an executable %geogebra fold by
1072 setting its **output** property to **geogebra**:

```
1073 1: %mathpiper, output="geogebra"  
1074 2:  
1075 3: x^2;  
1076 4:  
1077 5: %/mathpiper
```

1078 Executing this fold will produce the following output:

```
1079 1: %mathpiper, output="geogebra"  
1080 2:  
1081 3: x^2;  
1082 4:  
1083 5: %/mathpiper  
1084 6:  
1085 7: %geogebra  
1086 8: Result: x^2  
1087 9: %/geogebra
```

1088 Executing the generated %**geogebra** fold will produce an %output fold which
1089 tells the user that GeoGebra was updated and it will also send the function to the
1090 GeoGebra plugin for plotting. Illustration 4 shows the plot that was displayed:



*Illustration 4: MathMathPiper Function
Plotted With GeoGebra*

1091 **10.3.3.2 Displaying MathPiper Expressions In Traditional Form With HotEqn**

1092 Reading mathematical expressions in text form is often difficult. Being able to
 1093 view these expressions in traditional form when needed is helpful and a
 1094 %mathpiper fold can be configured to do this by setting its output property to
 1095 **latex**. When the fold is executed, it will generate an executable **%hoteqn** fold
 1096 that contains a MathPiper expression which has been converted into a LaTeX
 1097 expression. The %hoteqn fold can then be executed to view the expression in
 1098 traditional form:

```

1099 1: %mathpiper,output="latex"
1100 2:
1101 3: ((2*x)*(x+3)*(x+4))/9;
1102 5:
1103 6: %/mathpiper
1104 7:
1105 8: %hoteqn
1106 9: Result: \frac{2 x \left( x + 3\right) \left( x + 4\right) }{9}
1107 1: %/hoteqn
1108 2:
1109 3: %output,preserve="false"
1110 4: HotEqn updated.
1111 5: %/output

```

$$\frac{2x(x+3)(x+4)}{9}$$

1112 **10.3.4 %output**

1113 %output folds simply displays text output that has been generated by a parent
 1114 fold. It is not executable and therefore it is not highlighted in light blue like
 1115 executable folds are.

1116 **10.3.5 %error**

1117 %error folds display error messages that have been sent by the software that
 1118 was executing the code in a fold.

1119 **10.3.6 %html**

1120 %html folds display HTML code in a floating window as shown in the following
 1121 example:

```

1122 1: %html,x_size="700",y_size="440"
1123 2:

```

```

1124 3: <html>
1125 4:   <h1 align="center">HTML Color Values</h1>
1126 5:   <table border="0" cellpadding="10" cellspacing="1" width="600">
1127 6:     <tr>
1128 7:       <th bgcolor="white" colspan="2"></th>
1129 8:       <th colspan="6">where blue=cc</th>
1130 9:     </tr>
1131 10:    <tr>
1132 11:      <th rowspan="6">where&nbsp;red=</th>
1133 12:      <th>ff</th>
1134 13:      <th bgcolor="#ff00cc">ff00cc</th>
1135 14:      <th bgcolor="#ff33cc">ff33cc</th>
1136 15:      <th bgcolor="#ff66cc">ff66cc</th>
1137 16:      <th bgcolor="#ff99cc">ff99cc</th>
1138 17:      <th bgcolor="#ffcccc">ffcccc</th>
1139 18:      <th bgcolor="#ffffcc">ffffcc</th>
1140 19:    </tr>
1141 20:    <tr>
1142 21:      <th>cc</th>
1143 22:      <th bgcolor="#cc00cc">cc00cc</th>
1144 23:      <th bgcolor="#cc33cc">cc33cc</th>
1145 24:      <th bgcolor="#cc66cc">cc66cc</th>
1146 25:      <th bgcolor="#cc99cc">cc99cc</th>
1147 26:      <th bgcolor="#cccccc">cccccc</th>
1148 27:      <th bgcolor="#ccffcc">ccffcc</th>
1149 28:    </tr>
1150 29:    <tr>
1151 30:      <th>99</th>
1152 31:      <th bgcolor="#9900cc">
1153 32:        <font color="#ffffff">9900cc</font>
1154 33:      </th>
1155 34:      <th bgcolor="#9933cc">9933cc</th>
1156 35:      <th bgcolor="#9966cc">9966cc</th>
1157 36:      <th bgcolor="#9999cc">9999cc</th>
1158 37:      <th bgcolor="#99cccc">99cccc</th>
1159 38:      <th bgcolor="#99ffcc">99ffcc</th>
1160 39:    </tr>
1161 40:    <tr>
1162 41:      <th>66</th>
1163 42:      <th bgcolor="#6600cc">
1164 43:        <font color="#ffffff">6600cc</font>
1165 44:      </th>
1166 45:      <th bgcolor="#6633cc">
1167 46:        <font color="#FFFFFF">6633cc</font>
1168 47:      </th>
1169 48:      <th bgcolor="#6666cc">6666cc</th>
1170 49:      <th bgcolor="#6699cc">6699cc</th>
1171 50:      <th bgcolor="#66cccc">66cccc</th>
1172 51:      <th bgcolor="#66ffcc">66ffcc</th>
1173 52:    </tr>
1174 53:    <tr>
1175 54:      <th colspan="1"></th>

```

```
1176 55:          <th>00</th>
1177 56:          <th>33</th>
1178 57:          <th>66</th>
1179 58:          <th>99</th>
1180 59:          <th>cc</th>
1181 60:          <th>ff</th>
1182 61:      </tr>
1183 62:      <tr>
1184 63:          <th colspan="2"></th>
1185 64:          <th colspan="4">where green=</th>
1186 65:      </tr>
1187 66:  </table>
1188 67:</html>
1189 68:
1190 69:%/html
1191 70:
1192 71:      %output,preserve="false"
1193 72:
1194 73:      %/output
1195 74:
```

1196 This code produces the following output:

HTML Color Values

where blue=cc						
ff	ff00cc	ff33cc	ff66cc	ff99cc	ffcccc	ffffcc
cc	cc00cc	cc33cc	cc66cc	cc99cc	cccccc	ccffcc
99	9900cc	9933cc	9966cc	9999cc	99cccc	99ffcc
66	6600cc	6633cc	6666cc	6699cc	66cccc	66ffcc
	00	33	66	99	cc	ff
where green=						

where red=

1197 The %html fold's **width** and **height** properties determine the size of the display
1198 window.

1199 **10.3.7 %beanshell**

1200 BeanShell (<http://beanshell.org>) is a scripting language that uses Java syntax.
1201 MathRider uses BeanShell as its primary customization language and %beanshell
1202 folds give MathRider worksheets full access to the internals of MathRider along
1203 with the functionality provided by plugins. %beanshell folds are an advanced
1204 topic that will be covered in later books.

1205 **10.4 Automatically Inserting Folds & Removing Unpreserved Folds**

1206 Typing the the top and bottom fold lines (for example: %mathpiper ...
1207 %/mathpiper) can be tedious and MathRider has a way to automatically insert
1208 them. Place the cursor on a line in a .mrw worksheet file where you would like a
1209 fold inserted and then **press the right mouse button**. A popup menu will be
1210 displayed which will allow you to have a fold automatically inserted into the
1211 worksheet at position of the cursor.

1212 This popup menu also has a menu item called "**Remove Unpreserved Folds**". If
1213 this menu item is selected, all folds which have a "**preserve="false"**" property
1214 will be removed.

11 MathPiper Programming Fundamentals

(Note: in this section it is assumed that the reader has read section [7. MathPiper: A Computer Algebra System For Beginners](#) .)

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

11.1 Values and Expressions

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

`3`

represents the number three, the value:

`0.5`

represents the number one half, and the value:

`"Mathematics is powerful!"`

represents an English sentence.

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

`3`

`2 + 3`

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

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

```
In> 2 + 3
```

```
Result> 5
```

11.2 Operators

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

The **subtraction** operator is `-`, the **multiplication** operator is `*`, `/` is the **division** operator, `%` is the **remainder** operator, and `^` is the **exponent**

1249 operator. MathPiper has more operators in addition to these and some of them
1250 will be covered later.

1251 The following examples show the $-$, $*$, $/$, $\%$, and $^$ operators being used:

1252 In> 5 - 2
1253 Result> 3

1254 In> 3*4
1255 Result> 12

1256 In> 30/3
1257 Result> 10

1258 In> 8%5
1259 Result> 3

1260 In> 2^3
1261 Result> 8

1262 The $-$ character can also be used to indicate a negative number:

1263 In> -3
1264 Result> -3

1265 Subtracting a negative number results in a positive number:

1266 In> - -3
1267 Result> 3

1268 In MathPiper, **operators** are symbols (or groups of symbols) which are
1269 implemented with **functions**. One can either call the function an operator
1270 represents directly or use the operator to call the function indirectly. However,
1271 using operators requires less typing and they often make a program easier to
1272 read.

1273 **11.3 Operator Precedence**

1274 When expressions contain more than 1 operator, MathPiper uses a set of rules
1275 called **operator precedence** to determine the order in which the operators are
1276 applied to the values in the expression. Operator precedence is also referred to
1277 as the **order of operations**. Operators with higher precedence are evaluated
1278 before operators with lower precedence. The following table shows a subset of
1279 MathPiper's operator precedence rules with higher precedence operators being
1280 placed higher in the table:

1281 $^$ Exponents are evaluated right to left.

1282 $*$, $\%$, $/$ Then multiplication, remainder, and division operations are evaluated

1283 left to right.

1284 +, - Finally, addition and subtraction are evaluated left to right.

1285 Lets manually apply these precedence rules to the multi-operator expression we
1286 used earlier. Here is the expression in source code form:

1287 $5 + 6 * 21 / 18 - 2^3$

1288 And here it is in traditional form:

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

1289 According to the precedence rules, this is the order in which MathPiper
1290 evaluates the operations in this expression:

1291 $5 + 6 * 21 / 18 - 2^3$

1292 $5 + 6 * 21 / 18 - 8$

1293 $5 + 126 / 18 - 8$

1294 $5 + 7 - 8$

1295 $12 - 8$

1296 4

1297 Starting with the first expression, MathPiper evaluates the $^$ operator first which
1298 results in the **8** in the expression below it. In the second expression, the $*$
1299 operator is executed next, and so on. The last expression shows that the final
1300 result after all of the operators have been evaluated is **4**.

1301 **11.4 Changing The Order Of Operations In An Expression**

1302 The default order of operations for an expression can be changed by grouping
1303 various parts of the expression within parentheses **()**. Parentheses force the
1304 code that is placed inside of them to be evaluated before any other operators are
1305 evaluated. For example, the expression $2 + 4 * 5$ evaluates to 22 using the
1306 default precedence rules:

1307 In> $2 + 4 * 5$

1308 Result> 22

1309 If parentheses are placed around $4 + 5$, however, the addition operator is forced
1310 to be evaluated before the multiplication operator and the result is 30:

1311 In> $(2 + 4) * 5$

1312 Result> 30

1313 Parentheses can also be nested and nested parentheses are evaluated from the
1314 most deeply nested parentheses outward:

```
1315 In> ((2 + 4)*3)*5
1316 Result> 90
```

1317 Since parentheses are evaluated before any other operators, they are placed at
1318 the top of the precedence table:

- 1319 () Parentheses are evaluated from the inside out.
- 1320 ^ Then exponents are evaluated right to left.
- 1321 *,%,/ Then multiplication, remainder, and division operations are evaluated
1322 left to right.
- 1323 +, - Finally, addition and subtraction are evaluated left to right.

1324 **11.5 Variables**

1325 As discussed in section [7.1.2.1](#), variables are symbols that can be associated with
1326 values. One way to create variables in MathPiper is through **assignment** and
1327 this consists of placing the name of a variable you would like to create on the left
1328 side of an assignment operator **:=** and an expression on the right side of this
1329 operator. When the expression returns a value, the value is assigned (or **bound**
1330 to) to the variable.

1331 In the following example, a variable called **box** is created and the number **7** is
1332 assigned to it:

```
1333 In> box := 7
1334 Result> 7
```

1335 Notice that the assignment operator returns the value that was bound to the
1336 variable as its result. If you want to see the value that the variable box (or any
1337 variable) has been bound to, simply evaluate it:

```
1338 In> box
1339 Result> 7
```

1340 If a variable has not been bound to a value yet, it will return itself as the result
1341 when it is evaluated:

```
1342 In> box2
1343 Result> box2
```

1344 MathPiper variables are **case sensitive**. This means that MathPiper takes into

1345 account the **case** of each letter in a variable name when it is deciding if two or
1346 more variable names are the same variable or not. For example, the variable
1347 name **Box** and the variable name **box** are not the same variable because the first
1348 variable name starts with an upper case 'B' and the second variable name starts
1349 with a lower case 'b'.

1350 Programs are able to have more than 1 variable and here is a more sophisticated
1351 example which uses 3 variables:

```
1352 a := 2  
1353 Result> 2
```

```
1354 b := 3  
1355 Result> 3
```

```
1356 a + b  
1357 Result> 5
```

```
1358 answer := a + b  
1359 Result> 5
```

```
1360 answer  
1361 Result> 5
```

1362 The part of an expression that is on the right side of an assignment operator is
1363 always evaluated first and the result is then assigned to the variable that is on
1364 the left side of the operator.

1365 **11.6 Functions & Function Names**

1366 In programming, **functions** are named blocks of code that can be executed one
1367 or more times by being **called** from other parts of the same program or called
1368 from other programs. Functions can have values passed to them from the calling
1369 code and they always return a value back to the calling code when they are
1370 finished executing. An example of a function is the **IsEven()** function which was
1371 discussed in an previous section.

1372 Functions are one way that MathPiper enables code to be reused. Most
1373 programming languages allow code to be reused in this way, although in other
1374 languages these named blocks of code are sometimes called **subroutines**,
1375 **procedures**, **methods**, etc.

1376 The functions that come with MathPiper have names which consist of either a
1377 single word (such as **Add()**) or multiple words that have been put together to
1378 form a compound word (such as **IsBound()**). All letters in the names of
1379 functions which come with MathPiper are lower case except the beginning letter
1380 in each word, which are upper case.

1381 **11.7 Functions That Produce Side Effects**

1382 Most functions are executed to obtain the results they produce but some
1383 functions are executed in order have them perform work that is not in the form
1384 of a result. Functions that perform work that is not in the form of a result are
1385 said to produce **side effects**. Side effects include many forms of work such as
1386 sending information to the user, opening files, and changing values in memory.

1387 When a function produces a side effect which sends information to the user, this
1388 information has the words **Side effects:** placed before it instead of the word
1389 **Result:**. The **Echo()** function is an example of a function that produces a side
1390 effect and it is covered in the following section.

1391 **11.7.1 The Echo() and Write() Functions**

1392 The Echo() and Write() functions both send information to the user and this is
1393 often referred to as "printing" in this document. It may also be called "echoing"
1394 and "writing".

1395 **11.7.1.1 Echo()**

1396 The **Echo()** function takes one expression (or multiple expressions separated by
1397 commas) evaluates each expression, and then prints the results as side effect
1398 output. The following examples illustrate this:

```
1399 In> Echo(1)
1400 Result> True
1401 Side Effects>
1402 1
```

1403 In this example, the number 1 was passed to the Echo() function, the number
1404 was evaluated (all numbers evaluate to themselves), and the result of the
1405 evaluation was then printed as a side effect. Notice that Echo() **also returned a**
1406 **result**. In MathPiper, all functions return a result but functions whose main
1407 purpose is to produce a side effect usually just return a result of **True** if the side
1408 effect succeeded or **False** if it failed. In this case, Echo() returned a result of
1409 **True** because it was able to successfully print a 1 as its side effect.

1410 The next example shows multiple expressions being sent to Echo() (notice that
1411 the expressions are separated by commas):

```
1412 In> Echo(1,1+2,2*3)
1413 Result> True
1414 Side Effects>
1415 1 3 6
```

1416 The expressions were each evaluated and their results were returned as side
1417 effect output.

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

```
1422 1:%mathpiper
1423 2:
1424 3:Echo(1);
1425 4:
1426 5:Echo(1+2);
1427 6:
1428 7:Echo(2*3);
1429 8:
1430 9:%/mathpiper
1431 10:
1432 11:    %output,preserve="false"
1433 12:    Result: True
1434 13:
1435 14:    Side effects:
1436 15:    1
1437 16:    3
1438 17:    6
1439 18:    %/output
```

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

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

```
1444 1:%mathpiper
1445 2:
1446 3:a := 1;
1447 4:b := 2;
1448 5:c := 3;
1449 6:
1450 7:%/mathpiper
1451 8:
1452 9:    %output,preserve="false"
1453 10:    Result: 3
1454 11:    %/output
```

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

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


```
1462 1:%mathpiper
1463 2:
1464 3:a := 1;
1465 4:Echo(a);
1466 5:
1467 6:b := 2;
1468 7:Echo(b);
1469 8:
1470 9:c := 3;
1471 10:Echo(c);
1472 11:
1473 12:%/mathpiper
1474 13:
1475 14: %output,preserve="false"
1476 15:     Result: True
1477 16:
1478 17:     Side effects:
1479 18:         1
1480 19:         2
1481 20:         3
1482 21: %/output
```

1483 11.7.1.2 Write()

1484 The **Write()** function is similar to the Echo() function except it does not
1485 automatically drop the display down to the next line after it finishes executing:

```
1486 1:%mathpiper
1487 2:
1488 3:Write(1);
1489 4:
1490 5:Write(1+2);
1491 6:
1492 7:Echo(2*3);
1493 8:
1494 9:%/mathpiper
1495 10:
1496 11: %output,preserve="false"
1497 12:     Result: True
1498 13:
1499 14:     Side effects:
1500 15:         1 3 6
1501 16: %/output
```

1502 Write() and Echo() have other differences than the one discussed here and more
1503 information about them can be found in the documentation for these functions.

1504 **11.8 Expressions Are Separated By Semicolons**

1505 In the previous sections, you may have noticed that all of the expressions that
1506 were executed inside of a **%mathpiper** fold had a semicolon (;) after them but
1507 the expressions executed in the **MathPiper console** did not have a semicolon
1508 after them. MathPiper actually requires that all expressions end with a
1509 semicolon, but one does not need to add a semicolon to an expression which is
1510 typed into the MathPiper console because the console adds it automatically when
1511 the expression is executed.

1512 All the previous code examples have had each of their expressions on a separate
1513 line, but multiple expressions can also be placed on a single line because the
1514 semicolons tell MathPiper where one expression ends and the next one begins:

```
1515 1:%mathpiper
1516 2:
1517 3:a := 1; Echo(a); b := 2; Echo(b); c := 3; Echo(c);
1518 4:
1519 5:%/mathpiper
1520 6:
1521 7:    %output,preserve="false"
1522 8:    Result: True
1523 9:
1524 10:    Side effects:
1525 11:    1
1526 12:    2
1527 13:    3
1528 14:    %/output
```

1529 The spaces that are in the code on line 2 of this example are used to make the
1530 code more readable. Any spaces that are present within any expressions or
1531 between them are ignored by MathPiper and if we removed the spaces from the
1532 previous code, the output remains the same:

```
1533 1:%mathpiper
1534 2:
1535 3:a:=1;Echo(a);b:=2;Echo(b);c:= 3;Echo(c);
1536 4:
1537 5:%/mathpiper
1538 6:
1539 7:    %output,preserve="false"
1540 8:    Result: True
1541 9:
1542 10:    Side effects:
1543 11:    1
1544 12:    2
1545 13:    3
1546 14:    %/output
```

1547 **11.9 Strings**

1548 A **string** is a **value** that is used to hold text-based information. The typical
1549 expression that is used to create a string consists of **text which is enclosed**
1550 **within double quotes**. Strings can be assigned to variables just like numbers
1551 can and strings can also be displayed using the Echo() function. The following
1552 program assigns a string value to the variable 'a' and then echos it to the user:

```
1553 1:%mathpiper
1554 2:
1555 3:a := "Hello, I am a string.";
1556 4:Echo(a);
1557 5:
1558 6:%/mathpiper
1559 7:
1560 8:    %output,preserve="false"
1561 9:    Result: True
1562 10:
1563 11:    Side effects:
1564 12:    Hello, I am a string.
1565 13:    %/output
```

1566 A useful aspect of using MathPiper inside of MathRider is that variables that are
1567 assigned inside of a **%mathpiper fold** are accessible inside of the **MathPiper**
1568 **console** and variables that are assigned inside of the **MathPiper console** are
1569 available inside of **%mathpiper folds**. For example, after the above fold is
1570 executed, the string that has been bound to variable 'a' can be displayed in the
1571 MathPiper console:

```
1572 In> a
1573 Result> "Hello, I am a string."
```

1574 Individual characters in a string can be accessed by placing the character's
1575 position inside of brackets [] after the variable it is assigned. A character's
1576 position is determined by its distance from the left side of the string, starting at
1577 1. For example, in the above string, 'H' is at position 1, 'e' is at position 2, etc.
1578 The following code shows individual characters in the above string being
1579 accessed:

```
1580 In> a[1]
1581 Result> "H"

1582 In> a[2]
1583 Result> "e"

1584 In> a[3]
1585 Result> "l"
```

```
1586 In> a[4]
1587 Result> "l"
```

```
1588 In> a[5]
1589 Result> "o"
```

1590 A range of characters in a string can be accessed by using the .. "range"
1591 operator:

```
1592 In> a[8 .. 11]
1593 Result> "I am"
```

1594 The .. operator is covered in section [11.17.3.1. The .. Range Operator](#).

1595 **11.10 Comments**

1596 Source code can often be difficult to understand and therefore all programming
1597 languages provide the ability for **comments** to be included in the code.

1598 Comments are used to explain what the code near them is doing and they are
1599 usually meant to be read by humans instead of being processed by a computer.
1600 Comments are ignored when the program is executed.

1601 There are two ways that MathPiper allows comments to be added to source code.
1602 The first way is by placing two forward slashes // to the left of any text that is
1603 meant to serve as a comment. The text from the slashes to the end of the line
1604 the slashes are on will be treated as a comment. Here is a program that contains
1605 comments which use slashes:

```
1606 1:%mathpiper
1607 2://This is a comment.
1608 3:
1609 4:x := 2; //Set the variable x equal to 2.
1610 5:
1611 6:
1612 7:%/mathpiper
1613 8:
1614 9:    %output,preserve="false"
1615 10:    Result: 2
1616 11:    %/output
```

1617 When this program is executed, any text that starts with slashes is ignored.

1618 The second way to add comments to a MathPiper program is by enclosing the
1619 comments inside of slash-asterisk/asterisk-slash symbols /* */. This option is
1620 useful when a comment is too large to fit on one line. Any text between these
1621 symbols is ignored by the computer. This program shows a longer comment
1622 which has been placed between these symbols:

```
1623 1:%mathpiper
1624 2:
1625 3:/*
1626 4: This is a longer comment and it uses
1627 5: more than one line. The following
1628 6: code assigns the number 3 to variable
1629 7: x and then returns it as a result.
1630 8:*/
1631 9:
1632 10:x := 3;
1633 11:
1634 12:%/mathpiper
1635 13:
1636 14: %output,preserve="false"
1637 15:     Result: 3
1638 16: %/output
```

1639 **11.11 Conditional Operators**

1640 A conditional operator is an operator that is used to compare two values.
1641 Expressions that contain conditional operators return a **boolean value** and a
1642 **boolean value** is one that can either be **True** or **False**. Table 2 shows the
1643 conditional operators that MathPiper uses:

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

Table 2: Conditional Operators

1644 The following examples show each of the conditional operators in Table 2 being
1645 used to compare values that have been assigned to variables **x** and **y**:

```
1646 1:%mathpiper
```

```
1647 2:
1648 2:// Example 1.
1649 3:x := 2;
1650 4:y := 3;
1651 5:
1652 6:Echo(x, "=", y, ":", x = y);
1653 7:Echo(x, "!= ", y, ":", x != y);
1654 8:Echo(x, "< ", y, ":", x < y);
1655 9:Echo(x, "<=", y, ":", x <= y);
1656 10:Echo(x, "> ", y, ":", x > y);
1657 11:Echo(x, ">=", y, ":", x >= y);
1658 12:
1659 13:%mathpiper
1660 14:
1661 15:    %output,preserve="false"
1662 16:    Result: True
1663 17:
1664 18:    Side effects:
1665 19:    2 = 3 :False
1666 20:    2 != 3 :True
1667 21:    2 < 3 :True
1668 22:    2 <= 3 :True
1669 23:    2 > 3 :False
1670 24:    2 >= 3 :False
1671 25:    %/output

1672 1:%mathpiper
1673 2:
1674 3:    // Example 2.
1675 4:    x := 2;
1676 5:    y := 2;
1677 6:
1678 7:    Echo(x, "=", y, ":", x = y);
1679 8:    Echo(x, "!= ", y, ":", x != y);
1680 9:    Echo(x, "< ", y, ":", x < y);
1681 10:    Echo(x, "<=", y, ":", x <= y);
1682 11:    Echo(x, "> ", y, ":", x > y);
1683 12:    Echo(x, ">=", y, ":", x >= y);
1684 13:
1685 14:%mathpiper
1686 15:
1687 16:    %output,preserve="false"
1688 17:    Result: True
1689 18:
1690 19:    Side effects:
1691 20:    2 = 2 :True
1692 21:    2 != 2 :False
1693 22:    2 < 2 :False
1694 23:    2 <= 2 :True
1695 24:    2 > 2 :False
```

```

1696 25:      2 >= 2 :True
1697 25:      %/output

1698 1:%mathpiper
1699 2:
1700 3:// Example 3.
1701 4:x := 3;
1702 5:y := 2;
1703 6:
1704 7:Echo(x, "=", y, ":", x = y);
1705 8:Echo(x, "!= ", y, ":", x != y);
1706 9:Echo(x, "< ", y, ":", x < y);
1707 10:Echo(x, "<= ", y, ":", x <= y);
1708 11:Echo(x, "> ", y, ":", x > y);
1709 12:Echo(x, ">= ", y, ":", x >= y);
1710 13:
1711 14:%/mathpiper
1712 15:
1713 16:      %output,preserve="false"
1714 17:      Result: True
1715 18:
1716 19:      Side effects:
1717 20:      3 = 2 :False
1718 21:      3 != 2 :True
1719 22:      3 < 2 :False
1720 23:      3 <= 2 :False
1721 24:      3 > 2 :True
1722 25:      3 >= 2 :True
1723 26:      %/output

```

1724 Conditional operators are placed at a lower level of precedence than the other
 1725 operators we have covered to this point:

- 1726 () Parentheses are evaluated from the inside out.
- 1727 ^ Then exponents are evaluated right to left.
- 1728 *,%,/ Then multiplication, remainder, and division operations are evaluated
 1729 left to right.
- 1730 +, - Then addition and subtraction are evaluated left to right.
- 1731 =,!=,<,<=,>,>= Finally, conditional operators are evaluated.

1732 **11.12 Making Decisions With The If() Function & Predicate Expressions**

1733 All programming languages provide the ability to make decisions and the most
 1734 commonly used function for making decisions in MathPiper is the **If()** function.

1735 There are two calling formats for the If() function:

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

1736 A **predicate** is an expression which evaluates to either **True** or **False**. The way
 1737 the first form of the If() function works is that it evaluates the first expression in
 1738 its argument list (which is the "predicate" expression) and then looks at the value
 1739 that is returned. If this value is **True**, the "then" expression that is listed second
 1740 in the argument list is executed. If the predicate expression evaluates to **False**,
 1741 the "then" expression is not executed.

1742 The following program uses an If() function to determine if the number in
 1743 variable x is greater than 5. If x is greater than 5, the program will echo
 1744 "Greater" and then "End of program":

```
1745 1:%mathpiper
1746 2:
1747 3:x := 6;
1748 4:
1749 5:If(x > 5, Echo(x, "is greater than 5.));
1750 6:
1751 7:Echo("End of program.");
1752 8:
1753 9:%/mathpiper
1754 10:
1755 11:    %output,preserve="false"
1756 12:    Result: True
1757 13:
1758 14:    Side effects:
1759 15:    6 is greater than 5.
1760 16:    End of program.
1761 17:    %/output
```

1762 In this program, x has been set to 6 and therefore the expression $x > 5$ is **True**.
 1763 When the If() functions evaluates the predicate expression and determines it is
 1764 **True**, it then executes the Echo() function. The second Echo() function at the
 1765 bottom of the program prints "End of program" regardless of what the If()
 1766 function does.

1767 Here is the same program except that **x** has been set to **4** instead of **6**:

```
1768 1:%mathpiper
1769 2:
1770 3:x := 4;
1771 4:
1772 5:If(x > 5, Echo(x, "is greater than 5.));
1773 6:
1774 7:Echo("End of program.");
```



```

1775 8:
1776 9: %/mathpiper
1777 10:
1778 11:     %output,preserve="false"
1779 12:     Result: True
1780 13:
1781 14:     Side effects:
1782 15:     End of program.
1783 16: %/output

```

1784 This time the expression **x > 4** returns a value of **False** which causes the If()
 1785 function to not execute the "then" expression that was passed to it.

1786 The second form of the If() function takes a third "else" expression which is
 1787 executed only if the predicate expression is **False**. This program is similar to the
 1788 previous one except an "else" expression has been added to it:

```

1789 1: %mathpiper
1790 2:
1791 3: x := 4;
1792 4:
1793 5: If(x > 5, Echo(x, "is greater than 5."), Echo(x, "is NOT greater than 5.));
1794 6:
1795 7: Echo("End of program.");
1796 8:
1797 9: %/mathpiper
1798 10:
1799 11:     %output,preserve="false"
1800 12:     Result: True
1801 13:
1802 14:     Side effects:
1803 15:     4 is NOT greater than 5.
1804 16:     End of program.
1805 17: %/output

```

1806 **11.13 The And(), Or(), & Not() Boolean Functions & Infix Notation**

1807 **11.13.1 And()**

1808 Sometimes one needs to check if two or more expressions are all **True** and one
 1809 way to do this is with the **And()** function. The And() function has two calling
 1810 formats and this is the first one:

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

1811 This calling format is able to accept one or more expressions as input. If all of

1812 these expressions returns a value of **True**, the And() function will also return a
1813 **True**. However, if any of the expressions returns a **False**, then the And()
1814 function will return a **False**. This can be seen in the following examples:

1815 In> And(True, True)

1816 Result> True

1817 In> And(True, False)

1818 Result> False

1819 In> And(False, True)

1820 Result> False

1821 In> And(True, True, True, True)

1822 Result> True

1823 In> And(True, True, False, True)

1824 Result> False

1825 The second format (or **notation**) that can be used to call the And() function is
1826 called **infix** notation:

```
expression1 And expression2
```

1827 With **infix** notation, an expression is placed on both sides of the And() function
1828 name instead of being placed inside of parentheses that are next to it:

1829 In> True And True

1830 Result> True

1831 In> True And False

1832 Result> False

1833 In> False And True

1834 Result> False

1835 Infix notation can only accept two expressions at a time, but it is often more
1836 convenient to use than function calling notation. The following program
1837 demonstrates using the infix version of the And() function:

```
1838 1:%mathpiper
1839 2:
1840 3:a := 7;
1841 4:b := 9;
1842 5:
1843 6:Echo("1: ", a < 5 And b < 10);
1844 7:Echo("2: ", a > 5 And b > 10);
1845 8:Echo("3: ", a < 5 And b > 10);
```

```
1846 9:Echo("4: ", a > 5 And b < 10);
1847 10:
1848 11:If(a > 5 And b < 10, Echo("These expressions are both true.));
1849 12:
1850 13:%/mathpiper
1851 14:
1852 15:    %output,preserve="false"
1853 16:    Result: True
1854 17:
1855 18:    Side effects:
1856 19:    1: False
1857 20:    2: False
1858 21:    3: False
1859 22:    4: True
1860 23:    These expressions are both true.
1861 23:    %/output
```

1862 11.13.2 Or()

1863 The Or() function is similar to the And() function in that it has both a function
1864 and an infix calling format and it only works with boolean values. However,
1865 instead of requiring that all expressions be **True** in order to return a **True**, Or()
1866 will return a **True** if **one or more expressions are True**.

1867 Here is the function calling format for Or():

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

1868 and these examples show Or() being used with this format:

```
1869 In> Or(True, False)
1870 Result> True

1871 In> Or(False, True)
1872 Result> True

1873 In> Or(False, False)
1874 Result> False

1875 In> Or(False, False, False, False)
1876 Result> False

1877 In> Or(False, True, False, False)
1878 Result> True
```

1879 The infix notation format for Or() is as follows:

```
expression1 Or expression2
```

1880 and these examples show this notation being used:

1881 In> True Or False

1882 Result> True

1883 In> False Or True

1884 Result> True

1885 In> False Or False

1886 Result> False

1887 The following program also demonstrates using the infix version of the Or()
1888 function:

```
1889 1:%mathpiper
1890 2:
1891 3:a := 7;
1892 4:b := 9;
1893 5:
1894 6:Echo("1: ", a < 5 Or b < 10);
1895 7:Echo("2: ", a > 5 Or b > 10);
1896 8:Echo("3: ", a > 5 Or b < 10);
1897 9:Echo("4: ", a < 5 Or b > 10);
1898 10:
1899 11:If(a < 5 Or b < 10,Echo("At least one of these expressions is true.));
1900 12:
1901 13:%/mathpiper
1902 14:
1903 15:    %output,preserve="false"
1904 16:    Result: True
1905 17:
1906 18:    Side effects:
1907 19:    1: True
1908 20:    2: True
1909 21:    3: True
1910 22:    4: False
1911 23:    At least one of these expressions is true.
1912 24:    %/output
```

1913 11.13.3 Not() & Prefix Notation

1914 The **Not()** function works with boolean expressions like the And() and Or()
1915 functions do, except it can only accept one expression as input. The way Not()
1916 works is that it changes a **True** value to a **False** value and a **False** value to a
1917 **True** value. Here is the Not() function's normal calling format:

```
Not(expression)
```

1918 and these examples show Not() being used with this format:

1919 In> Not(True)

1920 Result> False

1921 In> Not(False)

1922 Result> True

1923 Instead of providing an alternative infix calling format like And() and Or() do,
1924 Not()'s second calling format uses **prefix** notation:

```
Not expression
```

1925 Prefix notation looks similar to function notation except no parentheses are used:

1926 In> Not True

1927 Result> False

1928 In> Not False

1929 Result> True

1930 Finally, here is a program that uses the prefix version of Not():

```
1931 1:%mathpiper
1932 2:
1933 3:Echo("3 = 3 is ", 3 = 3);
1934 4:
1935 5:Echo("Not 3 = 3 is ", Not 3 = 3);
1936 6:
1937 7:%/mathpiper
1938 8:
1939 9:    %output,preserve="false"
1940 10:    Result: True
1941 11:
1942 12:    Side effects:
1943 13:    3 = 3 is True
1944 14:    Not 3 = 3 is False
1945 15:    %/output
```

1946 11.14 The While() Looping Function & Bodied Notation

1947 Many kinds of machines, including computers, derive much of their power from
1948 the principle of **repeated cycling**. **Repeated cycling** in a program means to

1949 execute one or more expressions over and over again and this process is called
1950 "**looping**". MathPiper provides a number of ways to implement loops in a
1951 program and these ways range from straight-forward to subtle.

1952 We will begin discussing looping in MathPiper by starting with the straight-
1953 forward **While** function. The calling format for the **While** function is as follows:

```
1954 While(predicate)  
1955 [  
1956     body_expressions  
1957 ];
```

1958 The **While** function is similar to the **If** function except it will repeatedly execute
1959 the statements it contains as long as its "predicate" expression is **True**. As soon
1960 as the predicate expression returns a **False**, the While() function skips the
1961 expressions it contains and execution continues with the expression that
1962 immediately follows the While() function (if there is one).

1963 The expressions which are contained in a While() function are called its "**body**"
1964 and all functions which have body expressions are called "**bodied**" functions. If
1965 a body contains more than one expression then these expressions need to be
1966 placed within **brackets** []. What body expressions are will become clearer after
1967 looking at some example programs.

1968 The following program uses a While() function to print the integers from 1 to 10:

```
1969 1:%mathpiper  
1970 2:  
1971 3:// This program prints the integers from 1 to 10.  
1972 4:  
1973 5:  
1974 6:/*  
1975 7:    Initialize the variable x to 1  
1976 8:    outside of the While "loop".  
1977 9:*/  
1978 10:x := 1;  
1979 11:  
1980 12:While(x <= 10)  
1981 13:[  
1982 14:    Echo(x);  
1983 15:  
1984 16:    x := x + 1; //Increment x by 1.  
1985 17:];  
1986 18:  
1987 19:%/mathpiper  
1988 20:  
1989 21:%output,preserve="false"  
1990 22:    Result: True  
1991 23:  
1992 24:    Side effects:  
1993 25:    1
```

```
1994 26:      2
1995 27:      3
1996 28:      4
1997 29:      5
1998 30:      6
1999 31:      7
2000 32:      8
2001 33:      9
2002 34:     10
2003 35:    %/output
```

2004 In this program, a single variable called **x** is created. It is used to tell the Echo()
2005 function which **integer** to print and it is also used in the expression that
2006 determines if the While() function should continue to "**loop**" or not.

2007 When the program is executed, 1 is placed into **x** and then the While() function is
2008 called. The predicate expression **x** <= 10 becomes 1 <= 10 and, since 1 is less
2009 than or equal to 10, a value of **True** is returned by the expression.

2010 The While() function sees that the expression returned a **True** and therefore it
2011 executes all of the expressions inside of its **body** from top to bottom.

2012 The Echo() function prints the current contents of x (which is 1) and then the
2013 expression x := x + 1; is executed.

2014 The expression **x := x + 1;** is a standard expression form that is used in many
2015 programming languages. Each time an expression in this form is evaluated, it
2016 increases the variable it contains by 1. Another way to describe the effect this
2017 expression has on **x** is to say that it **increments x** by 1.

2018 In this case **x** contains 1 and, after the expression is evaluated, **x** contains 2.

2019 After the last expression inside of a While() function is executed, the While()
2020 function reevaluates its predicate expression to determine whether it should
2021 continue looping or not. Since **x** is 2 at this point, the predicate expression
2022 returns **True** and the code inside the body of the While() function is executed
2023 again. This loop will be repeated until **x** is incremented to 11 and the predicate
2024 expression returns **False**.

2025 The previous program can be adjusted in a number of ways to achieve different
2026 results. For example, the following program prints the integers from 1 to 100 by
2027 changing the 10 in the predicate expression to 100. A Write() function is used in
2028 this program so that its output is displayed on the same line until it encounters
2029 the wrap margin in MathRider (which can be set in Utilities -> Buffer Options...).

```
2030 1:%mathpiper
2031 2:
2032 3:// Print the integers from 1 to 100.
2033 4:
2034 5:x := 1;
2035 6:
2036 7:While(x <= 100)
```

```

2037 8:[
2038 9:    Write(x);
2039 10:
2040 11:    x := x + 1;  //Increment x by 1.
2041 12:];
2042 13:
2043 14:%mathpiper
2044 15:
2045 16:    %output,preserve="false"
2046 17:    Result: True
2047 18:
2048 19:    Side effects:
2049 20:    1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
2050    24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
2051    44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
2052    64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83
2053    84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
2054 21:    %/output

```

2055 The following program prints the odd integers from 1 to 99 by changing the
 2056 increment value in the increment expression from **1** to **2**:

```

2057 1:%mathpiper
2058 2:
2059 3://Print the odd integers from 1 to 99.
2060 4:
2061 5:x := 1;
2062 6:
2063 7:While(x <= 100)
2064 8:[
2065 9:    Write(x);
2066 10:    x := x + 2;  //Increment x by 2.
2067 11:];
2068 12:
2069 13:%mathpiper
2070 14:
2071 15:    %output,preserve="false"
2072 16:    Result: True
2073 17:
2074 18:    Side effects:
2075 19:    1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43
2076    45 47 49 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83
2077    85 87 89 91 93 95 97 99
2078 20:    %/output

```

2079 Finally, the following program prints the numbers from 1 to 100 in reverse order:

```

2080 1:%mathpiper
2081 2:
2082 3://Print the integers from 1 to 100 in reverse order.

```



```

2083 4:
2084 5:x := 100;
2085 6:
2086 7:While(x >= 1)
2087 8:[
2088 9:    Write(x);
2089 10:    x := x - 1; //Decrement x by 1.
2090 11:];
2091 12:
2092 13:%mathpiper
2093 14:
2094 15:    %output,preserve="false"
2095 16:    Result: True
2096 17:
2097 18:    Side effects:
2098 19:    100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82
2099    81 80 79 78 77 76 75 74 73 72 71 70 69 68 67 66 65 64 63
2100    62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44
2101    43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25
2102    24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4
2103    3 2 1
2104 20:    %/output

```

2105 In order to achieve the reverse ordering, this program had to initialize **x** to **100**,
 2106 check to see if **x** was **greater than or equal to 1** ($x \geq 1$), and **decrement** x by
 2107 **subtracting 1 from it** instead of adding 1 to it.

2108 **11.15 Long-Running Loops, Infinite Loops, & Interrupting Execution**

2109 It is easy to create a loop that will execute a **large number of times**, or even **an**
 2110 **infinite number of times**, either on purpose or by mistake. When you execute
 2111 a program that contains an **infinite loop**, it will run until you tell MathPiper to
 2112 **interrupt** its execution. This is done by selecting the **MathPiper Plugin** (which
 2113 has been placed near the upper left part of the application) and then pressing the
 2114 **"Stop Current Calculation"** button which it contains. (**Note: currently this**
 2115 **button only works if MathPiper is executed inside of a %mathpiper fold.**)

2116 Lets experiment with this button by executing a program that contains an infinite
 2117 loop and then stopping it:

```

2118 1:%mathpiper
2119 2:
2120 3://Infinite loop example program.
2121 4:
2122 5:x := 1;
2123 6:While(x < 10)
2124 7:[
2125 8:    answer := x + 1;
2126 9:];

```

```
2127 10:
2128 11: %/mathpiper
2129 12:
2130 13:     %output,preserve="false"
2131 14:     Processing...
2132 15:     %/output
```

2133 Since the contents of `x` is never changed inside the loop, the expression `x < 10`
2134 always evaluates to **True** which causes the loop to continue looping. Notice that
2135 the %output fold contains the word "**Processing...**" to indicate that the program
2136 is executing the code.

2137 Execute this program now and then interrupt it using the "**Stop Current**
2138 **Calculation**" button. When the program is interrupted, the %output fold will
2139 display the message "**User interrupted calculation**" to indicate that the
2140 program was interrupted.

2141 **11.16 Predicate Functions**

2142 A predicate function is a function that either returns **True** or **False**. Most
2143 predicate functions in MathPiper have their names begin with "Is". For example,
2144 `IsEven()`, `IsOdd()`, `IsInteger`, etc. The following examples show some of the
2145 predicate functions that are in MathPiper:

```
2146 In> IsEven(4)
2147 Result> True
```

```
2148 In> IsEven(5)
2149 Result> False
```

```
2150 In> IsZero(0)
2151 Result> True
```

```
2152 In> IsZero(1)
2153 Result> False
```

```
2154 In> IsNegativeInteger(-1)
2155 Result> True
```

```
2156 In> IsNegativeInteger(1)
2157 Result> False
```

```
2158 In> IsPrime(7)
2159 Result> True
```

```
2160 In> IsPrime(100)
2161 Result> False
```

2162 There is also an `IsBound()` and an `IsUnbound()` function that can be used to
2163 determine whether or not a value is bound to a given variable:

```
2164 In> a
2165 Result> a

2166 In> IsBound(a)
2167 Result> False

2168 In> a := 1
2169 Result> 1

2170 In> IsBound(a)
2171 Result> True

2172 In> Clear(a)
2173 Result> True

2174 In> a
2175 Result> a

2176 In> IsBound(a)
2177 Result> False
```

2178 **11.17 Lists: Values That Hold Sequences Of Expressions**

2179 The **list** value type is designed to hold expressions in an ordered collection or
2180 sequence. Lists are very flexible and they are one of the most heavily used value
2181 types in MathPiper. Lists can hold expressions of any type, they can grow and
2182 shrink as needed, and they can be nested. Expressions in a list can be accessed
2183 by their position in the list and they can also be replaced by other expressions.

2184 One way to create a list is by placing zero or more objects or expressions inside
2185 of a pair of **braces {}**. The following program creates a list that contains
2186 various expressions and assigns it to the variable x:

```
2187 In> x := {7,42,"Hello",1/2,var}
2188 Result> {7,42,"Hello",1/2,var}

2189 In> x
2190 Result> {7,42,"Hello",1/2,var}
```

2191 The number of expressions in a list can be determined with the **Length()**
2192 function:

```
2193 In> Length({7,42,"Hello",1/2,var})
2194 Result> 5
```

2195 A single expression in a list can be accessed by placing a set of **brackets []** to

2196 the right of the variable and then putting the expression's position number inside
2197 of the brackets (Notice that the first expression in the list is at position 1
2198 counting from the left side of the list):

```
2199 In> x[1]
2200 Result> 7
```

```
2201 In> x[2]
2202 Result> 42
```

```
2203 In> x[3]
2204 Result> "Hello"
```

```
2205 In> x[4]
2206 Result> 1/2
```

```
2207 In> x[5]
2208 Result> var
```

2209 The **1st** and **2nd** expressions in this list are **integers**, the **3rd** expression is a
2210 **string**, the **4th** expression is a **rational number** and the **5th** expression is a
2211 **variable**. Lists can also hold other lists as shown in the following example:

```
2212 In> x := {20, 30, {31, 32, 33}, 40}
2213 Result> {20,30,{31,32,33},40}
```

```
2214 In> x[1]
2215 Result> 20
```

```
2216 In> x[2]
2217 Result> 30
```

```
2218 In> x[3]
2219 Result> {31,32,33}
```

```
2220 In> x[4]
2221 Result> 40
2222
```

2223 The expression in the **3rd** position in the list is another **list** which contains the
2224 expressions **31**, **32**, and **33**. An expression in this second list can be accessed by
2225 two two sets of brackets:

```
2226 In> x[3][2]
2227 Result> 32
```

2228 The **3** inside of the first set of brackets accesses the **3rd** member of the **first** list
2229 and the **2** inside of the second set of brackets accesses the **2nd** member of the
2230 **second** list.

2231 11.17.1 Using While() Loops With Lists

2232 Functions that loop can be used to select each expression in a list in turn so that
2233 an operation can be performed on these expressions. The following program
2234 uses a While() loop to print each of the expressions in a list:

```
2235 1:%mathpiper
2236 2:
2237 3://Print each in in the list.
2238 4:
2239 5:x := {55,93,40,21,7,24,15,14,82};
2240 6:y := 1;
2241 7:
2242 8:While(y <= 9)
2243 9:[
2244 10:    Echo(y, "- ", x[y]);
2245 11:    y := y + 1;
2246 12:];
2247 13:
2248 14:%/mathpiper
2249 15:
2250 16:    %output,preserve="false"
2251 17:    Result: True
2252 18:
2253 19:    Side effects:
2254 20:    1 - 55
2255 21:    2 - 93
2256 22:    3 - 40
2257 23:    4 - 21
2258 24:    5 - 7
2259 25:    6 - 24
2260 26:    7 - 15
2261 27:    8 - 14
2262 28:    9 - 82
2263 29:    %/output
```

2264 A **loop** can also be used to search through a list. The following program uses a
2265 **While()** function and an **If()** function to search through a list to see if it contains
2266 the number **53**. If 53 is found in the list, a message is printed:

```
2267 1:%mathpiper
2268 2:
2269 3://Determine if 53 is in the list.
2270 4:
2271 5:testList := {18,26,32,42,53,43,54,6,97,41};
2272 6:index := 1;
2273 7:
2274 8:While(index <= 10)
2275 9:[
2276 10:    If(testList[index] = 53,
```

```

2277 11:         Echo("53 was found in the list at position", index));
2278 12:
2279 13:     index := index + 1;
2280 14:];
2281 15:
2282 16:%/mathpiper
2283 17:
2284 18:     %output,preserve="false"
2285 19:     Result: True
2286 20:
2287 21:     Side effects:
2288 22:     53 was found in the list at position 5
2289 23: %/output

```

2290 When this program was executed, it determined that **53** was present in the list at
 2291 position **5**.

2292 11.17.2 The ForEach() Looping Function

2293 The **ForEach()** function uses a **loop** to index through a list like the While()
 2294 function does, but it is more flexible and automatic. ForEach() uses bodied
 2295 notation like the While() function does and here is its calling format:

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

2296 **ForEach()** selects each expression in a list in turn, assigns it to the passed-in
 2297 "variable", and then executes the expressions that are inside of "body".
 2298 Therefore, body is executed once for each expression in the list.

2299 This example shows how ForEach() can be used to print all of the items in a list:

```

2300 1:%mathpiper
2301 2:
2302 3://Print all values in a list.
2303 4:
2304 5:ForEach(x, {50,51,52,53,54,55,56,57,58,59})
2305 6:[
2306 7:     Echo(x);
2307 8:];
2308 9:
2309 10:%/mathpiper
2310 11:
2311 12:     %output,preserve="false"
2312 13:     Result: True
2313 14:
2314 15:     Side effects:
2315 16:     50
2316 17:     51

```

```
2317 18:      52
2318 19:      53
2319 20:      54
2320 21:      55
2321 22:      56
2322 23:      57
2323 24:      58
2324 25:      59
2325 26:      %/output
```

2326 **11.18 Functions & Operators Which Loop Internally To Process Lists**

2327 Looping is such a useful capability that MathPiper has many functions which
2328 loop internally. This section discusses a number of functions that use internal
2329 loops to process lists.

2330 **11.18.1 TableForm()**

```
TableForm(list)
```

2331 The TableForm() function prints the contents of a list in the form of a table. Each
2332 member in the list is printed on its own line and this makes the contents of the
2333 list easier to read:

```
2334 In> testList := {2,4,6,8,10,12,14,16,18,20}
2335 Result> {2,4,6,8,10,12,14,16,18,20}
```

```
2336 In> TableForm(testList)
2337 Result> True
2338 Side Effects>
2339 2
2340 4
2341 6
2342 8
2343 10
2344 12
2345 14
2346 16
2347 18
2348 20
```

2349 **11.18.2 The .. Range Operator**

```
first .. last
```

2350 One often needs to create a list of consecutive integers and the .. range operator

2351 can be used to do this. The first integer in the list is placed before the ..
2352 operator (with a space in between them) and the last integer in the list is placed
2353 after the .. operator. Here are some examples:

```
2354 In> 1 .. 10  
2355 Result> {1,2,3,4,5,6,7,8,9,10}
```

```
2356 In> 10 .. 1  
2357 Result> {10,9,8,7,6,5,4,3,2,1}
```

```
2358 In> -10 .. 10  
2359 Result> {-10,-9,-8,-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,8,9,10}
```

2360 As the examples show, the .. operator can generate lists of integers in ascending
2361 order and descending order. It can also generate lists that contain negative
2362 integers.

2363 11.18.3 Contains()

2364 The **Contains()** function searches a list to determine if it contains a given
2365 expression. If it finds the expression, it returns **True** and if it doesn't find the
2366 expression, it returns **False**. Here is the calling format for Contains():

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

2367 The following code shows Contains() being used to locate a number in a list:

```
2368 In> Contains({50,51,52,53,54,55,56,57,58,59}, 53)  
2369 Result> True
```

```
2370 In> Contains({50,51,52,53,54,55,56,57,58,59}, 75)  
2371 Result> False
```

2372 The **Not()** function can also be used with predicate functions like Contains() to
2373 change their results:

```
2374 In> Not Contains({50,51,52,53,54,55,56,57,58,59}, 75)  
2375 Result> True
```

2376 11.18.4 Find()

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

2377 The **Find()** function searches a list for the first occurrence of a given expression.

2378 If the expression is found, the numerical position of its first occurrence is
2379 returned and if it is not found, -1 is returned:

```
2380 In> Find({23, 15, 67, 98, 64}, 15)
2381 Result> 2
```

```
2382 In> Find({23, 15, 67, 98, 64}, 8)
2383 Result> -1
```

2384 11.18.5 Count()

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

2385 **Count()** determines the number of times a given expression occurs in a list:

```
2386 In> testList := {a,b,b,c,c,c,d,d,d,d,e,e,e,e,e}
2387 Result> {a,b,b,c,c,c,d,d,d,d,e,e,e,e,e}
```

```
2388 In> Count(testList, c)
2389 Result> 3
```

```
2390 In> Count(testList, e)
2391 Result> 5
```

```
2392 In> Count(testList, z)
2393 Result> 0
```

2394 11.18.6 Select()

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

2395 **Select()** returns a list that contains all the expressions in a list which make a
2396 given predicate return **True**:

```
2397 In> Select("IsPositiveInteger", {46,87,59,-27,11,86,-21,-58,-86,-52})
2398 Result> {46,87,59,11,86}
```

2399 In this example, notice that the **name** of the predicate function is passed to
2400 Select() in **double quotes**. There are other ways to pass a predicate function to
2401 Select() but these are covered in a later section.

2402 Here are some further examples which use the Select() function:

```
2403 In> Select("IsOdd", {16,14,82,92,33,74,99,67,65,52})
2404 Result> {33,99,67,65}
```

```
2405 In> Select("IsEven", {16,14,82,92,33,74,99,67,65,52})
2406 Result> {16,14,82,92,74,52}
```

```
2407 In> Select("IsPrime", 1 .. 75)
2408 Result> {2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73}
```

2409 Notice how the third example uses the `..` operator to automatically generate a list
2410 of consecutive integers from 1 to 75 for the `Select()` function to analyze.

2411 11.18.7 The `Nth()` Function & The `[]` Operator

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

2412 The **`Nth()`** function simply returns the expression which is at a given index in a
2413 list. This example shows the third expression in a list being obtained:

```
2414 In> testList := {a,b,c,d,e,f,g}
2415 Result> {a,b,c,d,e,f,g}
```

```
2416 In> Nth(testList, 3)
2417 Result> c
```

2418 As discussed earlier, the `[]` operator can also be used to obtain a single
2419 expression from a list:

```
2420 In> testList[3]
2421 Result> c
```

2422 The `[]` operator can even obtain a single expression directly from a list without
2423 needing to use a variable:

```
2424 In> {a,b,c,d,e,f,g}[3]
2425 Result> c
```

2426 11.18.8 `Append()` & Nondestructive List Operations

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

2427 The **`Append()`** function adds an expression to the end of a list:

```
2428 In> testList := {21,22,23}
2429 Result> {21,22,23}
```

```
2430 In> Append(testList, 24)
2431 Result> {21,22,23,24}
```

2432 However, instead of changing the **original** list, MathPiper creates a **copy** of the
2433 **original** list and appends the expression to the **copy**. This can be confirmed by
2434 evaluating the variable **testList** after the Append() function has been called:

```
2435 In> testList
2436 Result> {21,22,23}
```

2437 Notice that the list that is bound to **testList** was not modified by the Append()
2438 function. This is called a **nondestructive list operation** and most MathPiper
2439 functions that manipulate lists do so nondestructively. To have the changed list
2440 bound to the variable that it being used, the following technique can be
2441 employed:

```
2442 In> testList := {21,22,23}
2443 Result> {21,22,23}
```

```
2444 In> testList := Append(testList, 24)
2445 Result> {21,22,23,24}
```

```
2446 In> testList
2447 Result> {21,22,23,24}
```

2448 After this code has been executed, the modified list has indeed been bound to
2449 testList as desired.

2450 There are some functions, such as DestructiveAppend(), which **do** change the
2451 original list and most of them begin with the word "Destructive". These are
2452 called "destructive functions" and it is recommended that destructive functions
2453 should be used with care.

2454 11.18.9 The : Prepend Operator

```
expression : list
```

2455 The prepend operator is a colon : and it can be used to add an expression to the
2456 beginning of a list:

```
2457 In> testList := {b,c,d}
2458 Result> {b,c,d}
```

```
2459 In> testList := a:testList
2460 Result> {a,b,c,d}
```

2461 **11.18.10 Concat()**

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

2462 The Concat() function is short for "concatenate" which means to join together
2463 sequentially. It takes two or more lists and joins them together into a
2464 single larger list:

```
2465 In> Concat({a,b,c}, {1,2,3}, {x,y,z})  
2466 Result> {a,b,c,1,2,3,x,y,z}
```

2467 **11.18.11 Insert(), Delete(), & Replace()**

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

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

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

2468 **Insert()** inserts an expression into a list at a given index, **Delete()** deletes an
2469 expression from a list at a given index, and **Replace()** replaces an expression in
2470 a list at a given index with another expression:

```
2471 In> testList := {a,b,c,d,e,f,g}  
2472 Result> {a,b,c,d,e,f,g}  
  
2473 In> testList := Insert(testList, 4, 123)  
2474 Result> {a,b,c,123,d,e,f,g}  
  
2475 In> testList := Delete(testList, 4)  
2476 Result> {a,b,c,d,e,f,g}  
  
2477 In> testList := Replace(testList, 4, xxx)  
2478 Result> {a,b,c,xxx,e,f,g}
```

2479 **11.18.12 Take()**

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

2480 **Take()** obtains a sublist from the **beginning** of a list, the **end** of a list, or the
2481 **middle** of a list. The expressions in the list that are not taken are discarded.

2482 A **positive** integer passed to Take() indicates how many expressions should be
2483 taken from the **beginning** of a list:

```
2484 In> testList := {a,b,c,d,e,f,g}
2485 Result> {a,b,c,d,e,f,g}
```

```
2486 In> Take(testList, 3)
2487 Result> {a,b,c}
```

2488 A **negative** integer passed to Take() indicates how many expressions should be
2489 taken from the **end** of a list:

```
2490 In> Take(testList, -3)
2491 Result> {e,f,g}
```

2492 Finally, if a **two member list** is passed to Take() it indicates the **range** of
2493 expressions that should be taken from the **middle** of a list. The **first** value in the
2494 passed-in list specifies the **beginning** index of the range and the **second** value
2495 specifies its **end**:

```
2496 In> Take(testList, {3,5})
2497 Result> {c,d,e}
```

2498 11.18.13 Drop()

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

2499 **Drop()** does the opposite of Take() in that it **drops** expressions from the
2500 **beginning** of a list, the **end** of a list, or the **middle** of a list and **returns a list**
2501 **which contains the remaining expressions**.

2502 A **positive** integer passed to Drop() indicates how many expressions should be
2503 dropped from the **beginning** of a list:

```
2504 In> testList := {a,b,c,d,e,f,g}
2505 Result> {a,b,c,d,e,f,g}
```

```
2506 In> Drop(testList, 3)
2507 Result> {d,e,f,g}
```

2508 A **negative** integer passed to Drop() indicates how many expressions should be
2509 dropped from the **end** of a list:

```
2510 In> Drop(testList, -3)
2511 Result> {a,b,c,d}
```

2512 Finally, if a **two member list** is passed to Drop() it indicates the **range** of
2513 expressions that should be dropped from the **middle** of a list. The **first** value in
2514 the passed-in list specifies the **beginning** index of the range and the **second**
2515 value specifies its **end**:

```
2516 In> Drop(testList, {3,5})
2517 Result> {a,b,f,g}
```

2518 11.18.14 FillList()

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

2519 The FillList() function simply creates a list which is of size "length" and fills it
2520 with "length" copies of the given expression:

```
2521 In> FillList(a, 5)
2522 Result> {a,a,a,a,a}

2523 In> FillList(42,8)
2524 Result> {42,42,42,42,42,42,42,42}
```

2525 11.18.15 RemoveDuplicates()

```
RemoveDuplicates(list)
```

2526 **RemoveDuplicates()** removes any duplicate expressions that are contained in
2527 in a list:

```
2528 In> testList := {a,a,b,c,c,b,b,a,b,c,c}
2529 Result> {a,a,b,c,c,b,b,a,b,c,c}

2530 In> RemoveDuplicates(testList)
2531 Result> {a,b,c}
```

2532 **11.18.16 Reverse()**

```
Reverse(list)
```

2533 **Reverse()** reverses the order of the expressions in a list:

```
2534 In> testList := {a,b,c,d,e,f,g,h}
```

```
2535 Result> {a,b,c,d,e,f,g,h}
```

```
2536 In> Reverse(testList)
```

```
2537 Result> {h,g,f,e,d,c,b,a}
```

2538 **11.18.17 Partition()**

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

2539 The **Partition()** function breaks a list into sublists of size "partition_size":

```
2540 In> testList := {a,b,c,d,e,f,g,h}
```

```
2541 Result> {a,b,c,d,e,f,g,h}
```

```
2542 In> Partition(testList, 2)
```

```
2543 Result> {{a,b},{c,d},{e,f},{g,h}}
```

2544 If the `partition_size` does not divide the length of the list evenly, the remaining
2545 elements are discarded:

```
2546 In> Partition(testList, 3)
```

```
2547 Result> {{h,b,c},{d,e,f}}
```

2548 The number of elements that `Partition()` will discard can be calculated by
2549 dividing the length of a list by the partition size and obtaining the remainder:

```
2550 In> Mod(Length(testList), 3)
```

```
2551 Result> 2
```

2552 The `Mod()` function, which divides two integers and return their remainder, is
2553 covered in a later section.

2554 **11.19 Functions That Work With Integers**

2555 This section discusses various functions which work with integers. Some of
2556 these functions also work with non-integer values and their use with non-
2557 integers is discussed in other sections.

2558 **11.19.1 RandomIntegerVector()**

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

2559 A vector can be thought of as a list that does not contain other lists.

2560 **RandomIntegerVector()** creates a list of size "length" that contains random
2561 integers that are no lower than "lowest_possible" and no higher than "highest
2562 possible". The following example creates **10** random integers between **1** and **99**
2563 inclusive:

```
2564 In> RandomIntegerVector(10, 1, 99)  
2565 Result> {73,93,80,37,55,93,40,21,7,24}
```

2566 **11.19.2 Max() & Min()**

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

2567 If two values are passed to Max(), it determines which one is larger:

```
2568 In> Max(10, 20)  
2569 Result> 20
```

2570 If a list of values are passed to Max(), it finds the largest value in the list:

```
2571 In> testList := RandomIntegerVector(10, 1, 99)  
2572 Result> {73,93,80,37,55,93,40,21,7,24}
```

```
2573 In> Max(testList)  
2574 Result> 93
```

2575 The **Min()** function is the opposite of the Max() function.

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

2576 If two values are passed to Min(), it determines which one is smaller:


```
2577 In> Min(10, 20)
2578 Result> 10
```

2579 If a list of values are passed to Min(), it finds the smallest value in the list:

```
2580 In> testList := RandomIntegerVector(10, 1, 99)
2581 Result> {73,93,80,37,55,93,40,21,7,24}
```

```
2582 In> Min(testList)
2583 Result> 7
```

2584 11.19.3 Div() & Mod()

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

2585 **Div()** stands for "divide" and determines the whole number of times a divisor
2586 goes into a dividend:

```
2587 In> Div(7, 3)
2588 Result> 2
```

2589 **Mod()** stands for "modulo" and it determines the remainder that results when a
2590 dividend is divided by a divisor:

```
2591 In> Mod(7,3)
2592 Result> 1
```

2593 The remainder/modulo operator % can also be used to calculate a remainder:

```
2594 In> 7 % 2
2595 Result> 1
```

2596 11.19.4 Gcd()

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

2597 GCD stands for Greatest Common Divisor and the **Gcd()** function determines the
2598 greatest common divisor of the values that are passed to it.

2599 If two integers are passed to Gcd(), it calculates their greatest common divisor:

```
2600 In> Gcd(21, 56)
2601 Result> 7
```

2602 If a list of integers are passed to Gcd(), it finds the greatest common divisor of all
2603 the integers in the list:

```
2604 In> Gcd({9, 66, 123})
2605 Result> 3
```

2606 11.19.5 Lcm()

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

2607 LCM stands for Least Common Multiple and the **Lcm()** function determines the
2608 least common multiple of the values that are passed to it.

2609 If two integers are passed to Lcm(), it calculates their least common multiple:

```
2610 In> Lcm(14, 8)
2611 Result> 56
```

2612 If a list of integers are passed to Lcm(), it finds the least common multiple of all
2613 the integers in the list:

```
2614 In> Lcm({3, 7, 9, 11})
2615 Result> 693
```

2616 11.19.6 Add()

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

2617 **Add()** can find the sum of two or values passed to it:

```
2618 In> Add(3, 8, 20, 11)
2619 Result> 42
```

2620 It can also find the sum of a list of values :

```
2621 In> testList := RandomIntegerVector(10, 1, 99)
2622 Result> {73, 93, 80, 37, 55, 93, 40, 21, 7, 24}
```

```
2623 In> Add(testList)
```

2624 `Result> 523`

2625 `In> testList := 1 .. 10`

2626 `Result> {1,2,3,4,5,6,7,8,9,10}`

2627 `In> Add(testList)`

2628 `Result> 55`

2629 **11.19.7 Factorize()**

```
Factorize(list)
```

2630 This function has two calling formats, only one of which is discussed here.

2631 **Factorize(list)** multiplies all the expressions in a list together and returns their
2632 product:

2633 `In> Factorize({1,2,3})`

2634 `Result> 6`

2635 **11.20 User Defined Functions**

2636 In computer programming, a **function** is a named sections of code that can be
2637 **called** from other sections of code. **Values** can be sent to a function for
2638 processing as part of the **call** and a function always returns a value as its result.

2639 The values that are sent to a function when it is called are called **arguments** and
2640 a function can accept 0 or more of them. These arguments are placed within
2641 parentheses.

2642 MathPiper has many predefined functions (some of which have been discussed in
2643 previous sections) but users can create their own functions too. The following
2644 program creates a function called **addNums()** which takes two numbers as
2645 arguments, adds them together, and returns their sum back to the calling code
2646 as a result:

2647 `In> addNums(num1,num2) := num1 + num2`

2648 `Result> True`

2649 This line of code defined a new function called **addNums** and specified that it
2650 will accept two values when it is called. The **first** value will be placed into the
2651 variable **num1** and the **second** value will be placed into the variable **num2**. The
2652 code on the **right side** of the assignment operator is then bound to this function
2653 and it is executed each time the function is called. The following example shows
2654 the new addNums() function being called multiple times with different values

2655 being passed to it:

2656 In> addNums(2,3)

2657 Result> 5

2658 In> addNums(4,5)

2659 Result> 9

2660 In> addNums(9,1)

2661 Result> 10

2662 Notice that, unlike the functions that come with MathPiper, we chose to have this
2663 function's name start with a **lower case letter**. We could have had addNums()
2664 begin with an upper case letter but it is a convention in MathPiper for user
2665 defined function names to begin with a lower case letter to distinguish them
2666 from the functions that come with MathPiper.

2667 The values that are returned from user defined functions can also be assigned to
2668 variables. The following example uses a %mathpiper fold to define a function
2669 called **evenIntegers()** and then this function is used in the MathPiper console:

```
2670 1:%mathpiper
2671 2:
2672 3:evenIntegers(endInteger) :=
2673 4:[
2674 5:    resultList := {};
2675 6:    x := 2;
2676 7:
2677 8:    While(x <= endInteger)
2678 9:    [
2679 10:        resultList := Append(resultList, x);
2680 11:        x := x + 2;
2681 12:    ];
2682 13:
2683 14:    resultList;
2684 15:];
2685 16:
2686 17:%/mathpiper
2687 18:
2688 19:    %output,preserve="false"
2689 20:        Result: True
2690 21:    %/output
```

2691 In> a := evenIntegers(10)

2692 Result> {2,4,6,8,10}

2693 In> Length(a)

2694 Result> 5

2695 The function evenIntegers() returns a list which contains all the even integers

2696 from 2 up through the value that was passed into it. The fold was first executed
2697 in order to define the evenIntegers() function and make it ready for use. The
2698 evenIntegers() function was then called from the MathPiper console and 10 was
2699 passed to it. After the function was finished executing, it return a list of even
2700 integers as a result and this result was assigned to the variable 'a'. We then
2701 passed the list that was assigned to 'a' to the Length() function in order to
2702 determine its size.

2703 11.20.1 Global Variables, Local Variables, & Local()

2704 The new evenIntegers() function seems to work well, but there is a problem. The
2705 variables 'x' and resultList were defined inside the function as **global variables**
2706 which means they are accessible from anywhere, including from within other
2707 functions, within folds:

```
2708 1: %mathpiper
2709 2:
2710 3: Echo(x, ",", resultList);
2711 4:
2712 5: %/mathpiper
2713 6:
2714 7:     %output,preserve="false"
2715 8:     Result: True
2716 9:
2717 10:     Side effects:
2718 11:         12 , {2,4,6,8,10}
2719 12: %/output
```

2720 and from within the MathPiper console:

```
2721 In> x
2722 Result> 12

2723 In> resultList
2724 Result> {2,4,6,8,10}
```

2725 Using global variables inside of functions is usually not a good idea because code
2726 in other functions and folds might already be using (or will use) the same
2727 variable names. Global variables which have the same name are the same
2728 variable. When one section of code changes the value of a given global variable,
2729 the value is changed everywhere that variable is used and this will eventually
2730 cause errors.

2731 In order to prevent errors like this, a function named **Local()** can be called
2732 inside a function to define what are called **local variables**. A **local variable** is
2733 only accessible inside the function it has been defined in, even if it has the same
2734 name as a global variable. The following example shows a second version of the

2735 evenIntegers() function which uses **Local()** to make **x** and **resultList** local
2736 variables:

```
2737 1:%mathpiper
2738 2:
2739 3:/*
2740 4: This version of evenIntegers() uses Local() to make
2741 5: x and resultList local variables
2742 6:*/
2743 7:
2744 8:evenIntegers(endInteger) :=
2745 9:[
2746 10:    Local(x,resultList);
2747 11:
2748 12:    resultList := {};
2749 13:    x := 2;
2750 14:
2751 15:    While(x <= endInteger)
2752 16:    [
2753 17:        resultList := Append(resultList, x);
2754 18:        x := x + 2;
2755 19:    ];
2756 20:
2757 21:    resultList;
2758 22:];
2759 23:
2760 24:%/mathpiper
2761 25:
2762 26:    %output,preserve="false"
2763 27:    Result: True
2764 28:    %/output
```

2765 We can verify that x and resultList are now local variables by first clearing them,
2766 calling evenIntegers(), and then seeing what x and resultList contain:

```
2767 In> Clear(x, resultList)
2768 Result> True

2769 In> evenIntegers(10)
2770 Result> {2,4,6,8,10}

2771 In> x
2772 Result> x

2773 In> resultList
2774 Result> resultList
```

2775 **11.21 Applying Functions To List Members**

2776 **11.21.1 Table()**

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

2777 The Table() function creates a list of values by doing the following:

- 2778 1) Generating a sequence of values between a "begin_value" and an
2779 "end_value" with each value being incremented by the "step_amount".
- 2780 2) Placing each value in the sequence into the specified "variable", one value
2781 at a time.
- 2782 3) Evaluating the defined "expression" (which contains the defined "variable")
2783 for each value, one at a time.
- 2784 4) Placing the result of each "expression" evaluation into the result list.

2785 This example generates a list which contains the integers 1 through 10:

```
2786 In> Table(x, x, 1, 10, 1)  
2787 Result> {1,2,3,4,5,6,7,8,9,10}
```

2788 Notice that the expression in this example is simply the variable itself with no
2789 other operations performed on it.

2790 The following example is similar to the previous one except that its expression
2791 multiplies x by 2:

```
2792 In> Table(x*2, x, 1, 10, 1)  
2793 Result> {2,4,6,8,10,12,14,16,18,20}
```

2794 Lists which contain decimal values can also be created by setting the
2795 "step_amount" to a decimal:

```
2796 In> Table(x, x, 0, 1, .1)  
2797 Result> {0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1}
```

2798 **12 THE CONTENT BELOW THIS LINE IS STILL UNDER**
2799 **DEVELOPMENT**

2800 **12.1 Sets**

2801 The following example shows operations that MathPiper can perform on sets:

```
2802 a = Set([0,1,2,3,4])
2803 b = Set([5,6,7,8,9,0])
2804 a,b
2805 |
2806 ({0, 1, 2, 3, 4}, {0, 5, 6, 7, 8, 9})
```

```
2807 a.cardinality()
2808 |
2809 5
```

```
2810 3 in a
2811 |
2812 True
```

```
2813 3 in b
2814 |
2815 False
```

```
2816 a.union(b)
2817 |
2818 {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

```
2819 a.intersection(b)
2820 |
2821 {0}
```


2822 **13 Miscellaneous Topics**

2823 **13.1 Errors**

2824 **13.2 Style Guide For Expressions**

2825 Always surround the following binary operators with a single space on either
2826 side: assignment ':=', comparisons (==, <, >, !=, <>, <=, >=, Booleans (and, or,
2827 not).

2828 Use spaces around the + and – arithmetic operators and no spaces around the
2829 *, /, %, and ^ arithmetic operators:

2830 $x = x + 1$

2831 $x = x*3 - 5\%2$

2832 $c = (a + b)/(a - b)$

2833 **13.3 Built-in Constants**

2834 MathPiper has a number of mathematical constants built into it and the following
2835 is a list of some of the more common ones:

2836 Pi, pi: The ratio of the circumference to the diameter of a circle.

2837 E, e: Base of the natural logarithm.

2838 I, i: The imaginary unit quantity.

2839

2840 log2: The natural logarithm of the real number 2.

2841 Infinity, infinity: Can have + or – placed before it to indicate positive or negative
2842 infinity.

2843 **14 Solving Equations**

2844 **14.1 Solving Equations Symbolically**

2845 **14.1.1 Symbolic Expressions & Simplify()**

2846 Expressions that contain symbolic variables are called symbolic expressions. In
2847 the following example, b is defined to be a symbolic variable and then it is used
2848 to create the symbolic expression $2*b$:

```
2849 var('b')
2850 type(2*b)
2851 |
2852 <class 'sage.calculus.calculus.SymbolicArithmetic'>
```

2853 As can be seen by this example, the symbolic expression $2*b$ was placed into an
2854 object of type SymbolicArithmetic. The expression can also be assigned to a
2855 variable:

```
2856 m = 2*b
2857 type(m)
2858 |
2859 <class 'sage.calculus.calculus.SymbolicArithmetic'>
```

2860 The following program creates two symbolic expressions, assigns them to
2861 variables, and then performs operations on them:

```
2862 m = 2*b
2863 n = 3*b
2864 m+n, m-n, m*n, m/n
2865 |
2866 (5*b, -b, 6*b^2, 2/3)
```

2867 Here is another example that multiplies two symbolic expressions together:

```
2868 m = 5 + b
2869 n = 8 + b
2870 y = m*n
2871 y
2872 |
2873 (b + 5)*(b + 8)
```

2874 **14.1.1.1 Expanding And Factoring**

2875 If the expanded form of the expression from the previous section is needed, it is
2876 easily obtained by calling the `expand()` method (this example assumes the cells in
2877 the previous section have been run):

```
2878 z = y.expand()
2879 z
2880 |
2881  $b^2 + 13b + 40$ 
2882 The expanded form of the expression has been assigned to variable z and the
2883 factored form can be obtained from z by using the factor() method:
```

```
2884 z.factor()
2885 |
2886  $(b + 5)(b + 8)$ 
2887 By the way, a number can be factored without being assigned to a variable by
2888 placing parentheses around it and calling its factor() method:
```

```
2889 (90).factor()
2890 |
2891  $2 * 3^2 * 5$ 
```

2892 **14.1.1.2 Miscellaneous Symbolic Expression Examples**

```
2893 var('a,b,c')
2894  $(5a + b + 4c) + (2a + 3b + c)$ 
2895 |
2896  $5c + 4b + 7a$ 
```

```
2897  $(a + b) - (x + 2b)$ 
2898 |
2899  $-x - b + a$ 
```

```
2900  $3a^2 - a(a - 5)$ 
2901 |
2902  $3a^2 - (a - 5)a$ 
```

```
2903  $\_.$ .factor()
2904 |
2905  $a(2a + 5)$ 
```

2906 **14.1.2 Symbolic Equations and The solve() Function**

```
2907 In addition to working with symbolic expressions, MathPiper is also able to work
2908 with symbolic equations:
```

```
2909 var('a')
2910  $\text{type}(x^2 == 16a^2)$ 
2911 |
```

2912 <class 'sage.calculus.equations.SymbolicEquation'>

2913 As can be seen by this example, the symbolic equation $x^2 == 16a^2$ was
2914 placed into an object of type SymbolicEquation. A symbolic equation needs to
2915 use double equals '==' so that it can be assigned to a variable using a single
2916 equals '=' like this:

2917 $m = x^2 == 16a^2$

2918 m , type(m)

2919 |

2920 ($x^2 == 16a^2$, <class 'sage.calculus.equations.SymbolicEquation'>)

2921 Many symbolic equations can be solved algebraically using the solve() function:

2922 solve(m , a)

2923 |

2924 [$a == -x/4$, $a == x/4$]

2925 The first parameter in the solve() function accepts a symbolic equation and the
2926 second parameter accepts the symbolic variable to be solved for.

2927 The solve() function can also solve simultaneous equations:

2928 var('i1,i2,i3,v0')

2929 $a = (i1 - i3)^2 + (i1 - i2)^5 + 10 - 25 == 0$

2930 $b = (i2 - i3)^3 + i2*1 - 10 + (i2 - i1)^5 == 0$

2931 $c = i3*14 + (i3 - i2)^3 + (i3 - i1)^2 - (-3*v0) == 0$

2932 $d = v0 == (i2 - i3)^3$

2933 solve([a,b,c,d], $i1,i2,i3,v0$)

2934 |

2935 [[$i1 == 4$, $i2 == 3$, $i3 == -1$, $v0 == 12$]]

2936 Notice that, when more than one equation is passed to solve(), they need to be
2937 placed into a list.

2938 **14.2 Solving Equations Numerically**

2939 **14.2.1 Roots**

2940 The sqrt() function can be used to obtain the square root of a value, but a more
2941 general technique is used to obtain other roots of a value. For example, if one
2942 wanted to obtain the cube root of 8:

2943 8 would be raised to the 1/3 power:

2944 $8^{(1/3)}$

2945 |

2946 2

2947 Due to the order of operations, the rational number 1/3 needs to be placed within
2948 parentheses in order for it to be evaluated as an exponent.

2949 **14.3 Finding Roots Graphically And Numerically With The `find_root()`**
2950 **Method**

2951 Sometimes equations cannot be solved algebraically and the `solve()` function
2952 indicates this by returning a copy of the input it was passed. This is shown in the
2953 following example:

```
2954 f(x) = sin(x) - x - pi/2
2955 eqn = (f == 0)
2956 solve(eqn, x)
2957 |
2958 [x == (2*sin(x) - pi)/2]
```

2959 However, equations that cannot be solved algebraically can be solved both
2960 graphically and numerically. The following example shows the above equation
2961 being solved graphically:

```
2962 show(plot(f,-10,10))
2963 |
```

2964 This graph indicates that the root for this equation is a little greater than -2.5.

2965 The following example shows the equation being solved more precisely using the
2966 `find_root()` method:

```
2967 f.find_root(-10,10)
2968 |
2969 -2.309881460010057
```

2970 The -10 and +10 that are passed to the `find_root()` method tell it the interval
2971 within which it should look for roots.

2972 **15 Output Forms**

2973 **15.1 LaTeX Is Used To Display Objects In Traditional Mathematics Form**

2974 LaTeX (pronounced lā-tek, <http://en.wikipedia.org/wiki/LaTeX>) is a document
2975 markup language which is able to work with a wide range of mathematical
2976 symbols. MathPiper objects will provide LaTeX descriptions of themselves when
2977 their latex() methods are called. The LaTeX description of an object can also be
2978 obtained by passing it to the latex() function:

2979 `a = (2*x^2)/7`

2980 `latex(a)`

2981 `|`

2982 `\frac{{2 \cdot {x^2}}}{7}`

2983 When this result is fed into LaTeX display software, it will generate traditional
2984 mathematics form output similar to the following:

2985 The jsMath package which is referenced in is the software that the MathPiper
2986 Notebook uses to translate LaTeX input into traditional mathematics form
2987 output.

2988 **15.2 Displaying Mathematical Objects In Traditional Form**

2989 Earlier it was indicated that MathPiper is able to display mathematical objects in
2990 either text form or traditional form. Up until this point, we have been using text
2991 form which is the default. If one wants to display a mathematical object in
2992 traditional form, the show() function can be used. The following example creates
2993 a mathematical expression and then displays it in both text form and traditional
2994 form:

2995 `var('y,b,c')`

2996 `z = (3*y^(2*b))/(4*x^c)^2`

2997 `#Display the expression in text form.`

2998 `z`

2999 `|`

3000 `3*y^(2*b)/(16*x^(2*c))`

3001 `#Display the expression in traditional form.`

3002 `show(z)`

3003 `|`

3004 **16 2D Plotting**

3005 **17 High School Math Problems (most of the problems are still in**
3006 **development)**

3007 **17.1 Pre-Algebra**

3008 Wikipedia entry.

3009 <http://en.wikipedia.org/wiki/Pre-algebra>

3010 (In development...)

3011 **17.1.1 Equations**

3012 Wikipedia entry.

3013 <http://en.wikipedia.org/wiki/Equation>

3014 (In development...)

3015 **17.1.2 Expressions**

3016 Wikipedia entry.

3017 http://en.wikipedia.org/wiki/Mathematical_expression

3018 (In development...)

3019 **17.1.3 Geometry**

3020 Wikipedia entry.

3021 <http://en.wikipedia.org/wiki/Geometry>

3022 (In development...)

3023 **17.1.4 Inequalities**

3024 Wikipedia entry.

3025 <http://en.wikipedia.org/wiki/Inequality>

3026 (In development...)

3027 **17.1.5 Linear Functions**

3028 Wikipedia entry.

3029 http://en.wikipedia.org/wiki/Linear_functions

3030 (In development...)

3031 **17.1.6 Measurement**

3032 Wikipedia entry.

3033 <http://en.wikipedia.org/wiki/Measurement>

3034 (In development...)

3035 **17.1.7 Nonlinear Functions**

3036 Wikipedia entry.

3037 http://en.wikipedia.org/wiki/Nonlinear_system

3038 (In development...)

3039 **17.1.8 Number Sense And Operations**

3040 Wikipedia entry.

3041 http://en.wikipedia.org/wiki/Number_sense

3042 Wikipedia entry.

3043 [http://en.wikipedia.org/wiki/Operation_\(mathematics\)](http://en.wikipedia.org/wiki/Operation_(mathematics))

3044 (In development...)

3045 **17.1.8.1 Express an integer fraction in lowest terms**

3046 """

3047 Problem:

3048 Express 90/105 in lowest terms.

3049 Solution:

3050 One way to solve this problem is to factor both the numerator and the
3051 denominator into prime factors, find the common factors, and then divide both
3052 the numerator and denominator by these factors.

3053 """

3054 n = 90

3055 d = 105

3056 print n,n.factor()

3057 print d,d.factor()

3058 |

3059 Numerator: 2 * 3² * 5

3060 Denominator: 3 * 5 * 7

3061 """

3062 It can be seen that the factors 3 and 5 each appear once in both the numerator
3063 and denominator, so we divide both the numerator and denominator by 3*5:

3064 """

3065 n2 = n/(3*5)

3066 d2 = d/(3*5)

3067 print "Numerator2:",n2

3068 print "Denominator2:",d2

3069 |

3070 Numerator2: 6

3071 Denominator2: 7

3072 """

3073 Therefore, 6/7 is 90/105 expressed in lowest terms.

3074 This problem could also have been solved more directly by simply entering
3075 90/105 into a cell because rational number objects are automatically reduced to
3076 lowest terms:
3077 ""
3078 90/105
3079 |
3080 6/7

3081 **17.1.9 Polynomial Functions**

3082 Wikipedia entry.
3083 http://en.wikipedia.org/wiki/Polynomial_function
3084 (In development...)

3085 **17.2 Algebra**

3086 Wikipedia entry.
3087 http://en.wikipedia.org/wiki/Algebra_1
3088 (In development...)

3089 **17.2.1 Absolute Value Functions**

3090 Wikipedia entry.
3091 http://en.wikipedia.org/wiki/Absolute_value
3092 (In development...)

3093 **17.2.2 Complex Numbers**

3094 Wikipedia entry.
3095 http://en.wikipedia.org/wiki/Complex_numbers
3096 (In development...)

3097 **17.2.3 Composite Functions**

3098 Wikipedia entry.
3099 http://en.wikipedia.org/wiki/Composite_function
3100 (In development...)

3101 **17.2.4 Conics**

3102 Wikipedia entry.
3103 <http://en.wikipedia.org/wiki/Conics>
3104 (In development...)

3105 **17.2.5 Data Analysis**

3106 Wikipedia entry.

3107 http://en.wikipedia.org/wiki/Data_analysis

3108 (In development...)

3109 **17.2.6 Discrete Mathematics**

3110 Wikipedia entry.

3111 http://en.wikipedia.org/wiki/Discrete_mathematics

3112 (In development...)

3113 **17.2.7 Equations**

3114 Wikipedia entry.

3115 <http://en.wikipedia.org/wiki/Equation>

3116 (In development...)

3117 **17.2.7.1 Express a symbolic fraction in lowest terms**

3118 """

3119 Problem:

3120 Express $(6x^2 - b) / (b - 6ab)$ in lowest terms, where a and b represent
3121 positive integers.

3122 Solution:

3123 """

3124 var('a,b')

3125 n = 6*a^2 - a

3126 d = b - 6 * a * b

3127 print n

3128 print "-----"

3129 print d

3130 |

3131
$$\frac{6a^2 - a}{b - 6ab}$$

3132

3133

3134

3135 """

3136 We begin by factoring both the numerator and the denominator and then looking
3137 for common factors:

3138 """

3139 n2 = n.factor()

3140 d2 = d.factor()

3141 print "Factored numerator:",n2.__repr__()

3142 print "Factored denominator:",d2.__repr__()

3143 |

3144 Factored numerator: a*(6*a - 1)

3145 Factored denominator: $-(6*a - 1)*b$

3146 """

3147 At first, it does not appear that the numerator and denominator contain any
 3148 common factors. If the denominator is studied further, however, it can be seen
 3149 that if $(1 - 6 a)$ is multiplied by -1 ,
 3150 $(6 a - 1)$ is the result and this factor is also present
 3151 in the numerator. Therefore, our next step is to multiply both the numerator and
 3152 denominator by -1 :

3153 """

3154 $n3 = n2 * -1$

3155 $d3 = d2 * -1$

3156 `print "Numerator * -1:",n3.__repr__()`

3157 `print "Denominator * -1:",d3.__repr__()`

3158 |

3159 Numerator * -1: $-a*(6*a - 1)$

3160 Denominator * -1: $(6*a - 1)*b$

3161 """

3162 Now, both the numerator and denominator can be divided by $(6*a - 1)$ in order to
 3163 reduce each to lowest terms:

3164 """

3165 `common_factor = 6*a - 1`

3166 `n4 = n3 / common_factor`

3167 `d4 = d3 / common_factor`

3168 `print n4`

3169 `print " ---"`

3170 `print d4`

3171 |

3172 $- a$

3173 $---$

3174 b

3175 """

3176 The problem could also have been solved more directly using a
 3177 SymbolicArithmetic object:

3178 """

3179 `z = n/d`

3180 `z.simplify_rational()`

3181 |

3182 $-a/b$

3183 **17.2.7.2 Determine the product of two symbolic fractions**

3184 Perform the indicated operation:

```
3185 """
3186 Since symbolic expressions are usually automatically simplified, all that needs to
3187 be done with this problem is to enter the expression and assign it to a variable:
3188 """

3189 var('y')
3190 a = (x/(2*y))^2 * ((4*y^2)/(3*x))^3

3191 #Display the expression in text form:
3192 a
3193 |
3194 16*y^4/(27*x)

3195 #Display the expression in traditional form:
3196 show(a)
3197 |

3198 17.2.7.3 Solve a linear equation for x
3199 Solve

3200 """
3201 Like terms will automatically be combined when this equation is placed into a
3202 SymbolicEquation object:
3203 """
3204 a = 5*x + 2*x - 8 == 5*x - 3*x + 7
3205 a
3206 |
3207 7*x - 8 == 2*x + 7
3208 """
3209 First, lets move the x terms to the left side of the equation by subtracting 2x
3210 from each side. (Note: remember that the underscore '_' holds the result of the
3211 last cell that was executed:
3212 """
3213 _ - 2*x
3214 |
3215 5*x - 8 == 7
3216 """
3217 Next, add 8 to both sides:
3218 """
3219 _ +8
```

```
3220 |
3221 5*x == 15
3222 """
3223 Finally, divide both sides by 5 to determine the solution:
3224 """
3225 _/5
3226 |
3227 x == 3
3228 """
3229 This problem could also have been solved automatically using the solve()
3230 function:
3231 """
3232 solve(a,x)
3233 |
3234 [x == 3]
```

3235 **17.2.7.4 Solve a linear equation which has fractions**

3236 Solve

```
3237 """
3238 The first step is to place the equation into a SymbolicEquation object. It is good
3239 idea to then display the equation so that you can verify that it was entered
3240 correctly:
3241 """
3242 a = (16*x - 13)/6 == (3*x + 5)/2 - (4 - x)/3
3243 a
3244 |
3245 (16*x - 13)/6 == (3*x + 5)/2 - (4 - x)/3
```

```
3246 """
3247 In this case, it is difficult to see if this equation has been entered correctly when
3248 it is displayed in text form so lets also display it in traditional form:
```

```
3249 """
3250 show(a)
3251 |
```

```
3252 """
3253 The next step is to determine the least common denominator (LCD) of the
3254 fractions in this equation so the fractions can be removed:
3255 """
3256 lcm([6,2,3])
3257 |
3258 6
```

3259 """

3260 The LCD of this equation is 6 so multiplying it by 6 removes the fractions:

3261 """

3262 $b = a*6$

3263 b

3264 $|$

3265 $16*x - 13 == 6*((3*x + 5)/2 - (4 - x)/3)$

3266 """

3267 The right side of this equation is still in factored form so expand it:

3268 """

3269 $c = b.expand()$

3270 c

3271 $|$

3272 $16*x - 13 == 11*x + 7$

3273 """

3274 Transpose the 11x to the left side of the equals sign by subtracting 11x from the

3275 SymbolicEquation:

3276 """

3277 $d = c - 11*x$

3278 d

3279 $|$

3280 $5*x - 13 == 7$

3281 """

3282 Transpose the -13 to the right side of the equals sign by adding 13 to the

3283 SymbolicEquation:

3284 """

3285 $e = d + 13$

3286 e

3287 $|$

3288 $5*x == 20$

3289 """

3290 Finally, dividing the SymbolicEquation by 5 will leave x by itself on the left side

3291 of the equals sign and produce the solution:

3292 """

3293 $f = e / 5$

3294 f

3295 $|$

3296 $x == 4$

3297 """

3298 This problem could have also be solved automatically using the solve() function:

3299 """

```
3300 solve(a,x)
3301 |
3302 [x == 4]
```

3303 **17.2.8 Exponential Functions**

3304 Wikipedia entry.
3305 http://en.wikipedia.org/wiki/Exponential_function
3306 (In development...)

3307 **17.2.9 Exponents**

3308 Wikipedia entry.
3309 <http://en.wikipedia.org/wiki/Exponent>
3310 (In development...)

3311 **17.2.10 Expressions**

3312 Wikipedia entry.
3313 [http://en.wikipedia.org/wiki/Expression_\(mathematics\)](http://en.wikipedia.org/wiki/Expression_(mathematics))
3314 (In development...)

3315 **17.2.11 Inequalities**

3316 Wikipedia entry.
3317 <http://en.wikipedia.org/wiki/Inequality>
3318 (In development...)

3319 **17.2.12 Inverse Functions**

3320 Wikipedia entry.
3321 http://en.wikipedia.org/wiki/Inverse_function
3322 (In development...)

3323 **17.2.13 Linear Equations And Functions**

3324 Wikipedia entry.
3325 http://en.wikipedia.org/wiki/Linear_functions
3326 (In development...)

3327 **17.2.14 Linear Programming**

3328 Wikipedia entry.
3329 http://en.wikipedia.org/wiki/Linear_programming
3330 (In development...)

3331 17.2.15 Logarithmic Functions

3332 Wikipedia entry.

3333 http://en.wikipedia.org/wiki/Logarithmic_function

3334 (In development...)

3335 17.2.16 Logistic Functions

3336 Wikipedia entry.

3337 http://en.wikipedia.org/wiki/Logistic_function

3338 (In development...)

3339 17.2.17 Matrices

3340 Wikipedia entry.

3341 [http://en.wikipedia.org/wiki/Matrix_\(mathematics\)](http://en.wikipedia.org/wiki/Matrix_(mathematics))

3342 (In development...)

3343 17.2.18 Parametric Equations

3344 Wikipedia entry.

3345 http://en.wikipedia.org/wiki/Parametric_equation

3346 (In development...)

3347 17.2.19 Piecewise Functions

3348 Wikipedia entry.

3349 http://en.wikipedia.org/wiki/Piecewise_function

3350 (In development...)

3351 17.2.20 Polynomial Functions

3352 Wikipedia entry.

3353 http://en.wikipedia.org/wiki/Polynomial_function

3354 (In development...)

3355 17.2.21 Power Functions

3356 Wikipedia entry.

3357 http://en.wikipedia.org/wiki/Power_function

3358 (In development...)

3359 17.2.22 Quadratic Functions

3360 Wikipedia entry.

3361 http://en.wikipedia.org/wiki/Quadratic_function

3362 (In development...)

3363 17.2.23 Radical Functions

3364 Wikipedia entry.

3365 http://en.wikipedia.org/wiki/Nth_root

3366 (In development...)

3367 17.2.24 Rational Functions

3368 Wikipedia entry.

3369 http://en.wikipedia.org/wiki/Rational_function

3370 (In development...)

3371 17.2.25 Sequences

3372 Wikipedia entry.

3373 <http://en.wikipedia.org/wiki/Sequence>

3374 (In development...)

3375 17.2.26 Series

3376 Wikipedia entry.

3377 http://en.wikipedia.org/wiki/Series_mathematics

3378 (In development...)

3379 17.2.27 Systems of Equations

3380 Wikipedia entry.

3381 http://en.wikipedia.org/wiki/System_of_equations

3382 (In development...)

3383 17.2.28 Transformations

3384 Wikipedia entry.

3385 [http://en.wikipedia.org/wiki/Transformation_\(geometry\)](http://en.wikipedia.org/wiki/Transformation_(geometry))

3386 (In development...)

3387 17.2.29 Trigonometric Functions

3388 Wikipedia entry.

3389 http://en.wikipedia.org/wiki/Trigonometric_function

3390 (In development...)

3391 17.3 Precalculus And Trigonometry

3392 Wikipedia entry.

3393 <http://en.wikipedia.org/wiki/Precalculus>

3394 <http://en.wikipedia.org/wiki/Trigonometry>

3395 (In development...)

3396 17.3.1 Binomial Theorem

3397 Wikipedia entry.

3398 http://en.wikipedia.org/wiki/Binomial_theorem

3399 (In development...)

3400 17.3.2 Complex Numbers

3401 Wikipedia entry.

3402 http://en.wikipedia.org/wiki/Complex_numbers

3403 (In development...)

3404 17.3.3 Composite Functions

3405 Wikipedia entry.

3406 http://en.wikipedia.org/wiki/Composite_function

3407 (In development...)

3408 17.3.4 Conics

3409 Wikipedia entry.

3410 <http://en.wikipedia.org/wiki/Conics>

3411 (In development...)

3412 17.3.5 Data Analysis

3413 Wikipedia entry.

3414 http://en.wikipedia.org/wiki/Data_analysis

3415 (In development...)

3416 17.3.6 Discrete Mathematics

3417 Wikipedia entry.

3418 http://en.wikipedia.org/wiki/Discrete_mathematics

3419 (In development...)

3420 17.3.7 Equations

3421 Wikipedia entry.

3422 <http://en.wikipedia.org/wiki/Equation>

3423 (In development...)

3424 17.3.8 Exponential Functions

3425 Wikipedia entry.

3426 <http://en.wikipedia.org/wiki/Equation>

3427 (In development...)

3428 17.3.9 Inverse Functions

3429 Wikipedia entry.

3430 http://en.wikipedia.org/wiki/Inverse_function

3431 (In development...)

3432 17.3.10 Logarithmic Functions

3433 Wikipedia entry.

3434 http://en.wikipedia.org/wiki/Logarithmic_function

3435 (In development...)

3436 17.3.11 Logistic Functions

3437 Wikipedia entry.

3438 http://en.wikipedia.org/wiki/Logistic_function

3439 (In development...)

3440 17.3.12 Mathematical Analysis

3441 Wikipedia entry.

3442 http://en.wikipedia.org/wiki/Mathematical_analysis

3443 (In development...)

3444 17.3.13 Matrices And Matrix Algebra

3445 Wikipedia entry.

3446 [http://en.wikipedia.org/wiki/Matrix_\(mathematics\)](http://en.wikipedia.org/wiki/Matrix_(mathematics))

3447 (In development...)

3448 17.3.14 Parametric Equations

3449 Wikipedia entry.

3450 http://en.wikipedia.org/wiki/Parametric_equation

3451 (In development...)

3452 17.3.15 Piecewise Functions

3453 Wikipedia entry.

3454 http://en.wikipedia.org/wiki/Piecewise_function

3455 (In development...)

3456 17.3.16 Polar Equations

3457 Wikipedia entry.

3458 http://en.wikipedia.org/wiki/Polar_equation

3459 (In development...)

3460 17.3.17 Polynomial Functions

3461 Wikipedia entry.

3462 http://en.wikipedia.org/wiki/Polynomial_function

3463 (In development...)

3464 17.3.18 Power Functions

3465 Wikipedia entry.

3466 http://en.wikipedia.org/wiki/Power_function

3467 (In development...)

3468 17.3.19 Quadratic Functions

3469 Wikipedia entry.

3470 http://en.wikipedia.org/wiki/Quadratic_function

3471 (In development...)

3472 17.3.20 Radical Functions

3473 Wikipedia entry.

3474 http://en.wikipedia.org/wiki/Nth_root

3475 (In development...)

3476 17.3.21 Rational Functions

3477 Wikipedia entry.

3478 http://en.wikipedia.org/wiki/Rational_function

3479 (In development...)

3480 17.3.22 Real Numbers

3481 Wikipedia entry.

3482 http://en.wikipedia.org/wiki/Real_number

3483 (In development...)

3484 17.3.23 Sequences

3485 Wikipedia entry.

3486 <http://en.wikipedia.org/wiki/Sequence>

3487 (In development...)

3488 17.3.24 Series

3489 Wikipedia entry.

3490 [http://en.wikipedia.org/wiki/Series_\(mathematics\)](http://en.wikipedia.org/wiki/Series_(mathematics))

3491 (In development...)

3492 17.3.25 Sets

3493 Wikipedia entry.

3494 <http://en.wikipedia.org/wiki/Set>

3495 (In development...)

3496 17.3.26 Systems of Equations

3497 Wikipedia entry.

3498 http://en.wikipedia.org/wiki/System_of_equations

3499 (In development...)

3500 17.3.27 Transformations

3501 Wikipedia entry.

3502 [http://en.wikipedia.org/wiki/Transformation_\(geometry\)](http://en.wikipedia.org/wiki/Transformation_(geometry))

3503 (In development...)

3504 17.3.28 Trigonometric Functions

3505 Wikipedia entry.

3506 http://en.wikipedia.org/wiki/Trigonometric_function

3507 (In development...)

3508 17.3.29 Vectors

3509 Wikipedia entry.

3510 <http://en.wikipedia.org/wiki/Vector>

3511 (In development...)

3512 17.4 Calculus

3513 Wikipedia entry.

3514 <http://en.wikipedia.org/wiki/Calculus>

3515 (In development...)

3516 17.4.1 Derivatives

3517 Wikipedia entry.

3518 <http://en.wikipedia.org/wiki/Derivative>

3519 (In development...)

3520 17.4.2 Integrals

3521 Wikipedia entry.

3522 <http://en.wikipedia.org/wiki/Integral>

3523 (In development...)

3524 17.4.3 Limits

3525 Wikipedia entry.

3526 [http://en.wikipedia.org/wiki/Limit_\(mathematics\)](http://en.wikipedia.org/wiki/Limit_(mathematics))

3527 (In development...)

3528 17.4.4 Polynomial Approximations And Series

3529 Wikipedia entry.

3530 http://en.wikipedia.org/wiki/Convergent_series

3531 (In development...)

3532 17.5 Statistics

3533 Wikipedia entry.

3534 <http://en.wikipedia.org/wiki/Statistics>

3535 (In development...)

3536 17.5.1 Data Analysis

3537 Wikipedia entry.

3538 http://en.wikipedia.org/wiki/Data_analysis

3539 (In development...)

3540 17.5.2 Inferential Statistics

3541 Wikipedia entry.

3542 http://en.wikipedia.org/wiki/Inferential_statistics

3543 (In development...)

3544 17.5.3 Normal Distributions

3545 Wikipedia entry.

3546 http://en.wikipedia.org/wiki/Normal_distribution

3547 (In development...)

3548 17.5.4 One Variable Analysis

3549 Wikipedia entry.

3550 <http://en.wikipedia.org/wiki/Univariate>

3551 (In development...)

3552 17.5.5 Probability And Simulation

3553 Wikipedia entry.

3554 <http://en.wikipedia.org/wiki/Probability>

3555 (In development...)

3556 **17.5.6 Two Variable Analysis**

3557 Wikipedia entry.

3558 <http://en.wikipedia.org/wiki/Multivariate>

3559 (In development...)

3560 **18 High School Science Problems**

3561 (In development...)

3562 **18.1 Physics**

3563 Wikipedia entry.

3564 <http://en.wikipedia.org/wiki/Physics>

3565 (In development...)

3566 **18.1.1 Atomic Physics**

3567 Wikipedia entry.

3568 http://en.wikipedia.org/wiki/Atomic_physics

3569 (In development...)

3570 **18.1.2 Boiling**

3571 Wikipedia entry.

3572 <http://en.wikipedia.org/wiki/Boiling>

3573 (In development...)

3574 **18.1.3 Buoyancy**

3575 Wikipedia entry.

3576 <http://en.wikipedia.org/wiki/Bouyancy>

3577 (In development...)

3578 **18.1.4 Circular Motion**

3579 Wikipedia entry.

3580 http://en.wikipedia.org/wiki/Circular_motion

3581 (In development...)

3582 **18.1.5 Convection**

3583 Wikipedia entry.

3584 <http://en.wikipedia.org/wiki/Convection>

3585 (In development...)

3586 **18.1.6 Density**

3587 Wikipedia entry.

3588 <http://en.wikipedia.org/wiki/Density>

3589 (In development...)

3590 18.1.7 Diffusion

3591 Wikipedia entry.

3592 <http://en.wikipedia.org/wiki/Diffusion>

3593 (In development...)

3594 18.1.8 Dynamics

3595 Wikipedia entry.

3596 [http://en.wikipedia.org/wiki/Dynamics_\(physics\)](http://en.wikipedia.org/wiki/Dynamics_(physics))

3597 (In development...)

3598 18.1.9 Electricity And Magnetism

3599 Wikipedia entry.

3600 <http://en.wikipedia.org/wiki/Electricity>

3601 <http://en.wikipedia.org/wiki/Magnetism>

3602 (In development...)

3603 18.1.10 Energy

3604 Wikipedia entry.

3605 <http://en.wikipedia.org/wiki/Energy>

3606 (In development...)

3607 18.1.11 Fluids

3608 Wikipedia entry.

3609 <http://en.wikipedia.org/wiki/Fluids>

3610 (In development...)

3611 18.1.12 Freezing

3612 Wikipedia entry.

3613 <http://en.wikipedia.org/wiki/Freezing>

3614 (In development...)

3615 18.1.13 Friction

3616 Wikipedia entry.

3617 <http://en.wikipedia.org/wiki/Friction>

3618 (In development...)

3619 18.1.14 Heat Transfer

3620 Wikipedia entry.

3621 http://en.wikipedia.org/wiki/Heat_transfer

3622 (In development...)

3623 18.1.15 Insulation

3624 Wikipedia entry.

3625 <http://en.wikipedia.org/wiki/Insulation>

3626 (In development...)

3627 18.1.16 Kinematics

3628 Wikipedia entry.

3629 <http://en.wikipedia.org/wiki/Kinematics>

3630 (In development...)

3631 18.1.17 Light

3632 Wikipedia entry.

3633 <http://en.wikipedia.org/wiki/Light>

3634 (In development...)

3635 18.1.18 Momentum

3636 Wikipedia entry.

3637 <http://en.wikipedia.org/wiki/Momentum>

3638 (In development...)

3639 18.1.19 Newton's Laws

3640 Wikipedia entry.

3641 http://en.wikipedia.org/wiki/Newtons_laws

3642 (In development...)

3643 18.1.20 Optics

3644 Wikipedia entry.

3645 <http://en.wikipedia.org/wiki/Optics>

3646 (In development...)

3647 18.1.21 Pressure

3648 Wikipedia entry.

3649 <http://en.wikipedia.org/wiki/Pressure>

3650 (In development...)

3651 18.1.22 Pulleys

3652 Wikipedia entry.

3653 <http://en.wikipedia.org/wiki/Pulley>

3654 (In development...)

3655 **18.1.23 Relativity**

3656 Wikipedia entry.

3657 <http://en.wikipedia.org/wiki/Relativity>

3658 (In development...)

3659 **18.1.24 Rotational Motion**

3660 Wikipedia entry.

3661 http://en.wikipedia.org/wiki/Rotational_motion

3662 (In development...)

3663 **18.1.25 Sound**

3664 Wikipedia entry.

3665 <http://en.wikipedia.org/wiki/Sound>

3666 (In development...)

3667 **18.1.26 Thermodynamics**

3668 Wikipedia entry.

3669 <http://en.wikipedia.org/wiki/Thermodynamics>

3670 (In development...)

3671 **18.1.27 Waves**

3672 Wikipedia entry.

3673 <http://en.wikipedia.org/wiki/Waves>

3674 (In development...)

3675 **18.1.28 Work**

3676 Wikipedia entry.

3677 http://en.wikipedia.org/wiki/Mechanical_work

3678 (In development...)