

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.....	19
4.3.2	Edit.....	19
4.3.3	Search.....	19
4.3.4	Markers.....	20
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.....	33
8.1 Function List.....	33
8.2 Mini Web Browser Interface.....	33
9 Using MathRider As A Programmer's Text Editor.....	35
9.1 Creating, Opening, And Saving Text Files.....	35
9.2 Editing Files.....	35
9.2.1 Rectangular Selection Mode.....	35
9.3 File Modes.....	36
9.4 Entering And Executing Stand Alone MathPiper Programs.....	36
10 MathRider Worksheet Files.....	37
10.1 Code Folds.....	37
10.2 Fold Properties.....	38
10.3 Currently Implemented Fold Types And Properties.....	39
10.3.1 %geogebra & %geogebra_xml.....	39
10.3.2 %hoteqn.....	42
10.3.3 %mathpiper.....	44
10.3.3.1 Plotting MathPiper Functions With GeoGebra.....	44
10.3.3.2 Displaying MathPiper Expressions In Traditional Form With HotEqn.....	45
10.3.4 %output.....	46
10.3.5 %error.....	46
10.3.6 %html.....	46

10.3.7 %beanshell.....	48
10.4 Automatically Inserting Folds & Removing Unpreserved Folds.....	48
11 MathPiper Programming Fundamentals.....	49
11.1 Values and Expressions.....	49
11.2 Operators.....	49
11.3 Operator Precedence.....	50
11.4 Changing The Order Of Operations In An Expression.....	51
11.5 Variables.....	52
11.6 Functions & Function Names.....	53
11.7 Functions That Produce Side Effects.....	54
11.7.1 The Echo() and Write() Functions.....	54
11.7.1.1 Echo().....	54
11.7.1.2 Write().....	56
11.8 Expressions Are Separated By Semicolons.....	57
11.9 Strings.....	58
11.10 Comments.....	59
11.11 Conditional Operators.....	60
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().....	66
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	76
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().....	83
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().....	86
11.18.17 Partition().....	87
11.19 Functions That Work With Integers.....	87
11.19.1 RandomIntegerVector().....	87
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.....	94
11.21.1 Table()	94
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.....	107
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.....	109
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.....	115
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.....	116
17.3.9 Inverse Functions.....	116
17.3.10 Logarithmic Functions.....	116
17.3.11 Logistic Functions.....	116
17.3.12 Matrices And Matrix Algebra.....	116
17.3.13 Mathematical Analysis.....	116
17.3.14 Parametric Equations.....	116
17.3.15 Piecewise Functions.....	117
17.3.16 Polar Equations.....	117
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.....	118
17.3.24 Series.....	118
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.....	119
17.4.2 Integrals.....	119
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.....	120
17.5.5 Probability And Simulation.....	120

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 Circular Motion.....	121
18.1.3 Dynamics.....	121
18.1.4 Electricity And Magnetism.....	121
18.1.5 Fluids.....	121
18.1.6 Kinematics.....	121
18.1.7 Light.....	122
18.1.8 Optics.....	122
18.1.9 Relativity.....	122
18.1.10 Rotational Motion.....	122
18.1.11 Sound.....	122
18.1.12 Waves.....	122
18.1.13 Thermodynamics.....	122
18.1.14 Work.....	122
18.1.15 Energy.....	123
18.1.16 Momentum.....	123
18.1.17 Boiling.....	123
18.1.18 Buoyancy.....	123
18.1.19 Convection.....	123
18.1.20 Density.....	123
18.1.21 Diffusion.....	123
18.1.22 Freezing.....	124
18.1.23 Friction.....	124
18.1.24 Heat Transfer.....	124
18.1.25 Insulation.....	124
18.1.26 Newton's Laws.....	124
18.1.27 Pressure.....	124
18.1.28 Pulleys.....	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 1.3 Support Email List

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

2 Introduction

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

2.1 What Is A Super Scientific Calculator?

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

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

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

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

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

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

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

and here is the same formula in text form:

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

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

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

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

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

73 ***2.2 What Is MathRider?***

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

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

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

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

91 **2.3 What Inspired The Creation Of Mathrider?**

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

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

95 and Steve Yegge's thoughts on learning mathematics:

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

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

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

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

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

108 **3 Downloading And Installing MathRider**

109 **3.1 *Installing Sun's Java Implementation***

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

115 **3.1.1 Installing Java On A Windows PC**

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

119 <http://java.com/>

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

122 **3.1.2 Installing Java On A Macintosh**

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

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

127 **3.1.3 Installing Java On A Linux PC**

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

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

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

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

```
java -version
```

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

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

3.2 Downloading And Extracting

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

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

<http://mathrider.org>

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

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

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

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

179 3.2.1 Extracting The Archive File For Windows Users

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

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

186 3.2.2 Extracting The Archive File For Unix Users

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

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

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

193 3.3 MathRider's Directory Structure & Execution Instructions

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

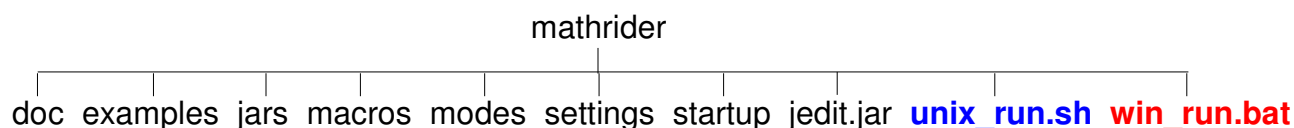


Illustration 1: MathRider's Directory Structure

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

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

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

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

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

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

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

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

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

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

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

209 **3.3.1 Executing MathRider On Windows Systems**

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

211 **3.3.2 Executing MathRider On Unix Systems**

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

214 `sh unix_run.sh`

215 **3.3.2.1 MacOS X**

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

219 `cd /Applications/mathrider`

220 Run mathrider by typing:

221 `sh unix_run.sh`

222 4 The Graphical User Interface

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

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

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

237 4.1 Buffers And Text Areas

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

244 4.2 The Gutter

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

248 4.3 Menus

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

255 4.3.1 File

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

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

261 4.3.2 Edit

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

267 4.3.3 Search

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

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

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

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

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

292 4.3.4 Markers

293 The Markers menu contains actions which place markers into a buffer, removes
294 them, and scrolls the document to them when they are selected. When a marker
295 is placed into a buffer, a link to it will be added to the bottom of the Markers
296 menu. Selecting a marker link will scroll the buffer to the marker it points to.
297 The list of marker links are kept in a temporary file which is placed into the same
298 directory as the buffer's file.

299 4.3.5 Folding

300 A **fold** is a section of a buffer that can be hidden (folded) or shown (unfolded) as
301 needed. In [worksheet files](#) (which have a .mrw extension) folds are created by
302 wrapping sections of a buffer in tags. For example, HTML folds start with a
303 %html tag and end with an %/html tag. See the **worksheet_demo_1.mws** file
304 for examples of folds.

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

307 4.3.6 View

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

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

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

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

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

325 4.3.7 Utilities

326 The utilities menu contains a significant number of actions, some that are useful
327 to beginners and others that are meant for experts. The two actions that are

328 most useful to beginners are the **Buffer Options** actions and the **Global**
329 **Options** actions. The **Buffer Options** actions allows the currently selected
330 buffer to be customized and the **Global Options** actions brings up a rich dialog
331 window that allows numerous aspects of the MathRider application to be
332 configured.
333 Feel free to explore these two actions in order to learn more about what they do.

334 4.3.8 Macros

335 **Macros** are small programs that perform useful tasks for the user. The top of
336 the **Macros** menu contains actions which allow macros to be created by
337 recording a sequence of user steps which can be saved for later execution. The
338 bottom of the **Macros** menu contains macros that can be executed as needed.
339 The main language that MathRider uses for macros is called **BeanShell** and it is
340 based upon Java's syntax. Significant parts of MathRider are written in
341 BeanShell, including many of the actions which are present in the menus. After
342 a user knows how to program in BeanShell, it can be used to easily customize
343 (and even extend) MathRider.

344 4.3.9 Plugins

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

348 4.3.10 Help

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

352 4.4 The Toolbar

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

360 5 MathRider's Plugin-Based Extension Mechanism

361 5.1 What Is A Plugin?

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

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

372 5.2 Which Plugins Are Currently Included When MathRider Is Installed?

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

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

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

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

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

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

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

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

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

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

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

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

395 **5.3 What Kinds Of Plugins Are Possible?**

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

398 **5.3.1 Plugins Based On Java Applets**

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

402 **5.3.2 Plugins Based On Java Applications**

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

404 **5.3.3 Plugins Which Talk To Native Applications**

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

408 6 Exploring The MathRider Application

409 6.1 The Console

410 The lower left window contains consoles. Switch to the MathPiper console by
411 pressing the small black inverted triangle which is near the word **System**.
412 Select the MathPiper console and when it comes up, enter simple **mathematical**
413 **expressions** (such as $2+2$ and $3*7$) and execute them by pressing **<enter>**.

414 6.2 MathPiper Program Files

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

418 6.3 MathRider Worksheets

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

424 6.4 Plugins

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

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

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

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

438 The Jung plugin was detached from the main window so it can be resized and
439 placed wherever it is needed. Select the inverted black triangle on the floating
440 windows and try docking the Jung plugin back to the main window again,
441 perhaps in a different position.

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

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

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

449 **7 MathPiper: A Computer Algebra System For Beginners**

450 Computer algebra system plugins are among the most exciting and powerful
451 plugins that can be used with MathRider. In fact, computer algebra systems are
452 so important that one of the reasons for creating MathRider was to provide a
453 vehicle for delivering a compute algebra system to as many people as possible.
454 If you like using a scientific calculator, you should love using a computer algebra
455 system!

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

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

468 **7.1 Numeric Vs. Symbolic Computations**

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

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

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

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

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.

525 In addition to addition, MathPiper can also do subtraction, multiplication,
526 exponents, and division:

527 In> 5-2
528 Result> 3

529 In> 3*4
530 Result> 12

531 In> 2^3
532 Result> 8

533 In> 12/6
534 Result> 2

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

539 MathPiper can also work with decimal numbers:

540 In> .5+1.2
541 Result> 1.7

542 In> 3.7-2.6
543 Result> 1.1

544 In> 2.2*3.9
545 Result> 8.58

546 In> 2.2^3
547 Result> 10.648

548 In> 9.5/3.2
549 Result> 9.5/3.2

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

553 In> N(9.5/3.2)
554 Result> 2.96875

555 7.1.1.1 Functions

556 **N()** is an example of a **function**. A function can be thought of as a "black box"
557 which accepts input, processes the input, and returns a result. Each function

558 has a name and in this case, the name of the function is **N** which stands for
559 **Numeric**. To the right of a function's name there is always a set of parentheses
560 and information that is sent to the function is placed inside of them. The purpose
561 of the N() function is to make sure that the information that is sent to it is
562 processed numerically instead of symbolically.

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

567 **7.1.1.2 Accessing Previous Input And Results**

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

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

```
574 In> 5*8  
575 Result> 40
```

```
576 In> %  
577 Result> 40
```

```
578 In> %*2  
579 Result> 80
```

580 **7.1.1.3 Syntax Errors**

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

```
586 In> 12 \ 6  
  
587 Error parsing expression, near token \
```

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

591 This section provided a short introduction to using MathPiper as a numeric

592 calculator and the next section contains a short introduction to using MathPiper
593 as a symbolic calculator.

594 7.1.2 Using The MathPiper Console As A Symbolic Calculator

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

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

```
600 In> 1/2 + 1/3  
601 Result> 5/6
```

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

```
606 In> %  
607 Result> 5/6
```

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

```
610 In> Numer(%)  
611 Result> 5
```

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

615 7.1.2.1 Variables

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

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

```
626 In> a := 1/2 + 1/3
627 Result> 5/6
```

```
628 In> a
629 Result> 5/6
```

```
630 In> Numer(a)
631 Result> 5
```

```
632 In> Denom(a)
633 Result> 6
```

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

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

```
642 In> a := 9
643 Result> 9
```

```
644 In> a
645 Result> 9
```

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

```
648 In> Clear(a)
649 Result> True
```

```
650 In> a
651 Result> a
```

652 Notice that the Clear() function returns '**True**' as a result after it is finished to
653 indicate that the variable that was sent to it was successfully cleared (or
654 **unbound**). Many functions either return '**True**' or '**False**' to indicate whether or

not the operation they performed succeeded. Also notice that unbound variables return themselves when they are evaluated. In this case, 'a' returned 'a'.

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

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

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

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

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

```
In> x
Result> x
```

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

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

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

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

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

Now that it has been shown how to use the MathPiper console as both a **symbolic** and a **numeric** calculator, we are ready to dig deeper into MathPiper. As you will soon discover, MathPiper contains an amazing number of functions which deal with a wide range of mathematics.

692 8 The MathPiper Documentation Plugin

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

698 8.1 Function List

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

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

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

722 8.2 Mini Web Browser Interface

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

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

733 **9 Using MathRider As A Programmer's Text Editor**

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

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

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

746 **9.1 Creating, Opening, And Saving Text Files**

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

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

759 **9.2 Editing Files**

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

765 **9.2.1 Rectangular Selection Mode**

766 One capability that MathRider has that a word process may not have is the
767 ability to select rectangular sections of text. To see how this works, do the
768 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.

808 10 MathRider Worksheet Files

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

816 10.1 Code Folds

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

```
823 1:%mathpiper
824 2:
825 3:"Hello World!";
826 4:
827 5:%/mathpiper
```

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

```
832 1:%mathpiper
833 2:
834 3:"Hello World!";
835 4:
836 5:%/mathpiper
837 6:
838 7:    %output,preserve="false"
839 8:    Result: "Hello World!"
840 9:    %/output
```

841 The default type of an output fold is **%output** and this one starts at **line 7** and
842 ends on **line 9**. Folds that can be executed have their first and last lines
843 **highlighted** and folds that cannot be executed do not have their first and last
844 lines highlighted. By default, folds of type **%output** have their **preserve**
845 **property** set to **false**. This tells MathRider to overwrite the **%output** fold with a

846 new version during the next execution of its parent.

847 **10.2 Fold Properties**

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

```

852 1:%mathpiper,output="pretty"
853 2:
854 3:x^2 + x/2 + 3;
855 4:
856 5:%/mathpiper
857 6:
858 7:    %output,preserve="false"
859 8:    Result: True
860 9:
861 10:    Side effects:
862 11:
863 12:        2    x
864 13:       x  + - + 3
865 14:          2
866 15:    %/output
  
```

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

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

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

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

```

879 1:%mathpiper,name="example_1",output="pretty"
880 2:
881 3:x^2 + x/2 + 3;
882 4:
883 5:%/mathpiper
  
```

```

884 6:
885 7:   %output,preserve="false"
886 8:   Result: True
887 9:
888 10:   Side effects:
889 11:
890 12:       2    x
891 13:     x  +  -  + 3
892 14:           2
893 15: %/output

```

894 10.3 Currently Implemented Fold Types And Properties

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

897 10.3.1 %geogebra & %geogebra_xml.

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

```

904 1: %geogebra,clear="true"
905 2:
906 3: //Plot a function.
907 4: f(x)=2*sin(x)
908 5:
909 6: //Add a tangent line to the function.
910 7: a = 2
911 8: (2,0)
912 9: t = Tangent[a, f]
913 10:
914 11: %/geogebra
915 12:
916 13:   %output,preserve="false"
917 14:   GeoGebra updated.
918 15:   %/output

```

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

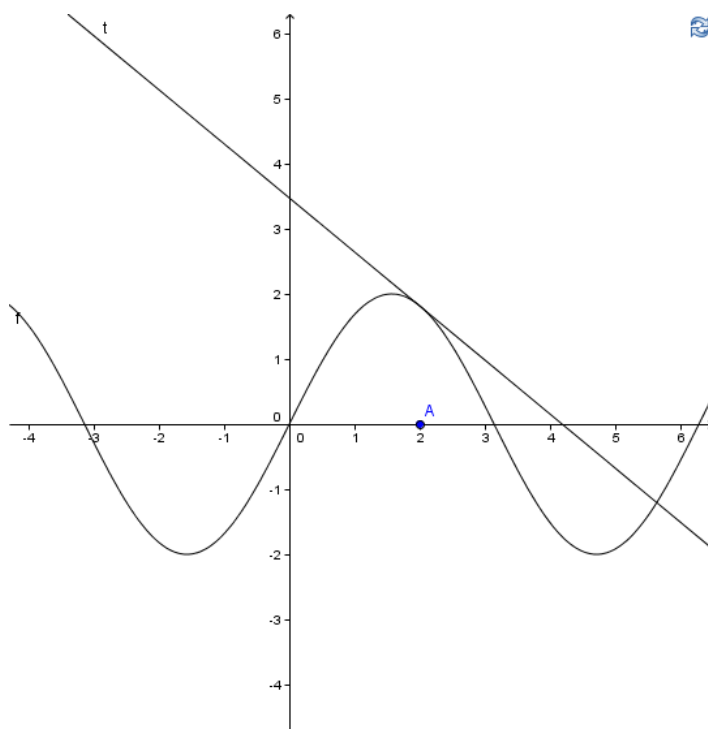


Illustration 2: GeoGebra: $\sin x$ and a tangent to it at $x=2$.

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

```

927 1: %geogebra_xml,description="Obtained from .ggb file"
928 2:
929 3: <?xml version="1.0" encoding="utf-8"?>
930 4: <geogebra format="3.0">
931 5: <gui>
932 6:   <show algebraView="true" auxiliaryObjects="true"
933   algebraInput="true" cmdList="true"/>
934 7:   <splitDivider loc="196" locVertical="400" horizontal="true"/>
935 8:   <font size="12"/>
936 9: </gui>
937 10: <euclidianView>
938 11:   <size width="540" height="553"/>
939 12:   <coordSystem xZero="215.0" yZero="315.0" scale="50.0"
940   yscale="50.0"/>
941 13:   <evSettings axes="true" grid="true" pointCapturing="3"
942   pointStyle="0" rightAngleStyle="1"/>
943 14:   <bgColor r="255" g="255" b="255"/>
944 15:   <axesColor r="0" g="0" b="0"/>

```



```

945 16:    <gridColor r="192" g="192" b="192"/>
946 17:    <lineStyle axes="1" grid="10"/>
947 18:    <axis id="0" show="true" label="" unitLabel="" tickStyle="1"
948    showNumbers="true"/>
949 19:    <axis id="1" show="true" label="" unitLabel="" tickStyle="1"
950    showNumbers="true"/>
951 20:    <grid distX="0.5" distY="0.5"/>
952 21: </euclidianView>
953 22: <kernel>
954 23:    <continuous val="true"/>
955 24:    <decimals val="2"/>
956 25:    <angleUnit val="degree"/>
957 26:    <coordStyle val="0"/>
958 27: </kernel>
959 28: <construction title="" author="" date="">
960 29: <expression label="f" exp="f(x) = 2 sin(x)"/>
961 30: <element type="function" label="f">
962 31:    <show object="true" label="true"/>
963 32:    <objColor r="0" g="0" b="255" alpha="0.0"/>
964 33:    <labelMode val="0"/>
965 34:    <animation step="0.1"/>
966 35:    <fixed val="false"/>
967 36:    <breakpoint val="false"/>
968 37:    <lineStyle thickness="2" type="0"/>
969 38: </element>
970 39: <element type="numeric" label="a">
971 40:    <value val="2.0"/>
972 41:    <show object="false" label="true"/>
973 42:    <objColor r="0" g="0" b="0" alpha="0.1"/>
974 43:    <labelMode val="1"/>
975 44:    <animation step="0.1"/>
976 45:    <fixed val="false"/>
977 46:    <breakpoint val="false"/>
978 47: </element>
979 48: <element type="point" label="A">
980 49:    <show object="true" label="true"/>
981 50:    <objColor r="0" g="0" b="255" alpha="0.0"/>
982 51:    <labelMode val="0"/>
983 52:    <animation step="0.1"/>
984 53:    <fixed val="false"/>
985 54:    <breakpoint val="false"/>
986 55:    <coords x="2.0" y="0.0" z="1.0"/>
987 56:    <coordStyle style="cartesian"/>
988 57:    <pointSize val="3"/>
989 58: </element>
990 59: <command name="Tangent">
991 60:    <input a0="a" a1="f"/>
992 61:    <output a0="t"/>
993 62: </command>
994 63: <element type="line" label="t">

```

```

995 64: <show object="true" label="true"/>
996 65: <objColor r="255" g="0" b="0" alpha="0.0"/>
997 66: <labelMode val="0"/>
998 67: <breakpoint val="false"/>
999 68: <coords x="0.8322936730942848" y="1.0" z="-3.4831821998399333"/>
1000 69: <lineStyle thickness="2" type="0"/>
1001 70: <eqnStyle style="explicit"/>
1002 71: </element>
1003 72: </construction>
1004 73: </geogebra>
1005 74:
1006 75: %/geogebra_xml
1007 76:
1008 77: %output,preserve="false"
1009 78: GeoGebra updated.
1010 79: %/output

```

1011 Illustration 3 shows the result of sending this XML code to GeoGebra:

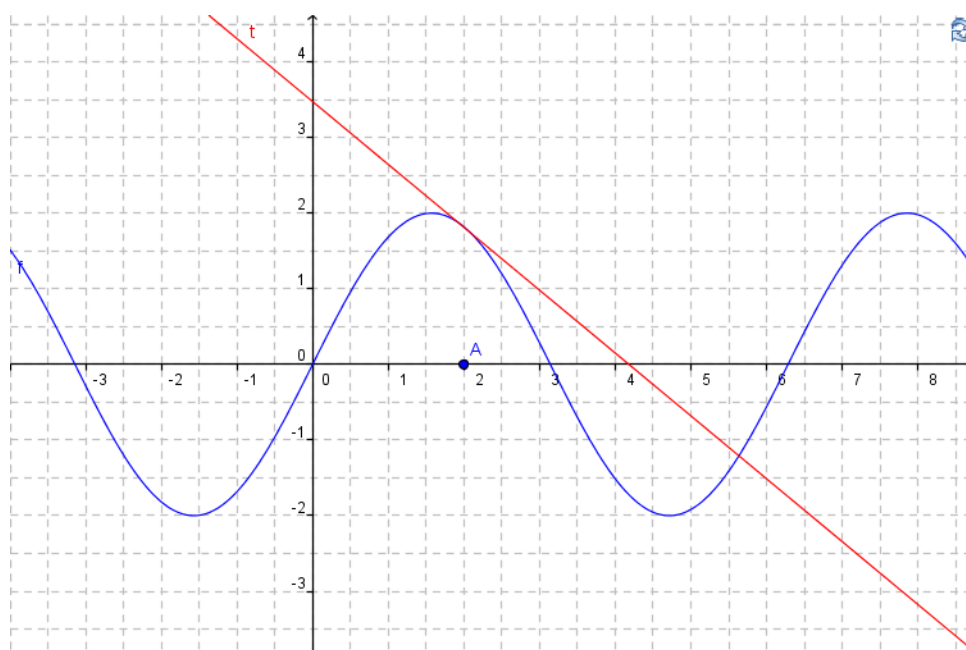


Illustration 3: Generated from %geogebra_xml fold.

1012 **%geogebra_xml** folds are not as easy to work with as plain **%geogebra** folds,
 1013 but they have the advantage of giving the user full control over the GeoGebra
 1014 environment. Both types of folds can be used together while working with
 1015 GeoGebra and this means that the user can send code to the GeoGebra plugin
 1016 from multiple folds during a work session.

1017 10.3.2 %hoteqn

1018 Before understanding what the HotEqn (<http://www.atp.ruhr-uni->

1019 bochum.de/VCLab/software/HotEqn/HotEqn.html) plugin does, one must first
 1020 know a little bit about LaTeX. LaTeX is a **markup language** which allows
 1021 formatting information (such as font size, color, and italics) to be added to plain
 1022 text. LaTeX was designed for creating technical documents and therefore it is
 1023 capable of marking up mathematics-related text. The hoteqn plugin accepts
 1024 input marked up with LaTeX's mathematics-oriented commands and displays it in
 1025 **traditional mathematics** form. For example, to have HotEqn show 2^3 , send it
 1026 `2^{3}`:

```

1027 1:%hoteqn
1028 2:
1029 3:2^{3}
1030 4:
1031 5:%/hoteqn
1032 6:
1033 7:    %output,preserve="false"
1034 8:    HotEqn updated.
1035 9:    %/output

```

1036 and it will display:

$$2^3$$

1037 To have HotEqn show $2x^3 + 14x^2 + \frac{24x}{7}$, send it the following code:

```

1038 1:%hoteqn
1039 2:
1040 3:2 x ^{3} + 14 x ^{2} + \frac{24 x}{7}
1041 4:
1042 5:%/hoteqn
1043 6:
1044 7:    %output,preserve="false"
1045 8:    HotEqn updated.
1046 9:    %/output

```

1047 and it will display:

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

1048 %hoteqn folds are handy for displaying typed-in LaTeX text in traditional form,
 1049 but their main use is to allow other folds to display mathematical objects in
 1050 traditional form. The next section discusses this second use further.

1051 **10.3.3 %mathpiper**

1052 %mathpiper folds were introduced in a previous section and later sections
1053 discuss how to start programming in MathPiper. This section shows how
1054 properties can be used to tell %mathpiper folds to generate output that can be
1055 sent to plugins.

1056 **10.3.3.1 Plotting MathPiper Functions With GeoGebra**

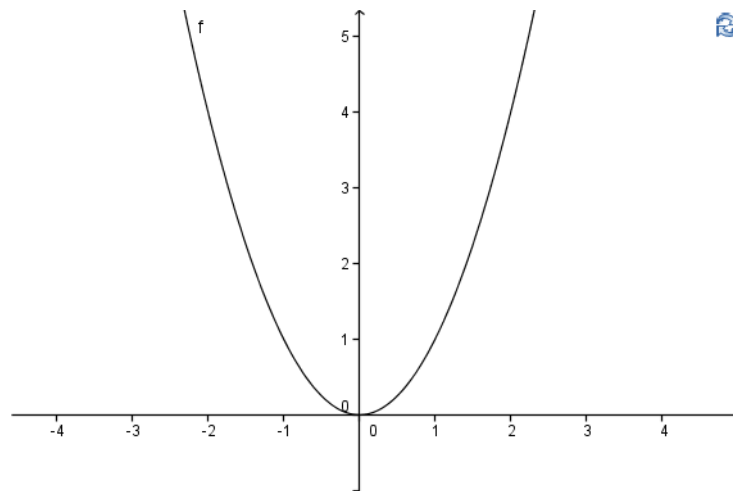
1057 When working with a computer algebra system, a user often needs to plot a
1058 function in order to understand it better. GeoGebra can plot functions and a
1059 %mathpiper fold can be configured to generate an executable %geogebra fold by
1060 setting its **output** property to **geogebra**:

```
1061 1:%mathpiper,output="geogebra"  
1062 2:  
1063 3:x^2;  
1064 4:  
1065 5:%/mathpiper
```

1066 Executing this fold will produce the following output:

```
1067 1:%mathpiper,output="geogebra"  
1068 2:  
1069 3:x^2;  
1070 4:  
1071 5:%/mathpiper  
1072 6:  
1073 7: %geogebra  
1074 8: Result: x^2  
1075 9: %/geogebra
```

1076 Executing the generated **%geogebra** fold will produce an %output fold which
1077 tells the user that GeoGebra was updated and it will also send the function to the
1078 GeoGebra plugin for plotting. Illustration 4 shows the plot that was displayed:



*Illustration 4: MathMathPiper Function
Plotted With GeoGebra*

1079 10.3.3.2 Displaying MathPiper Expressions In Traditional Form With HotEqn

1080 Reading mathematical expressions in text form is often difficult. Being able to
 1081 view these expressions in traditional form when needed is helpful and a
 1082 %mathpiper fold can be configured to do this by setting its output property to
 1083 **latex**. When the fold is executed, it will generate an executable %hoteqn fold
 1084 that contains a MathPiper expression which has been converted into a LaTeX
 1085 expression. The %hoteqn fold can then be executed to view the expression in
 1086 traditional form:

```

1087 1:%mathpiper,output="latex"
1088 2:
1089 3:((2*x)*(x+3)*(x+4))/9;
1090 5:
1091 6:%/mathpiper
1092 7:
1093 8:    %hoteqn
1094 9:    Result: \frac{2 x \left( x + 3\right) \left( x + 4\right) }{9}
1095 1:    %/hoteqn
1096 2:
1097 3:    %output,preserve="false"
1098 4:    HotEqn updated.
1099 5:    %/output
  
```

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

1100 10.3.4 %output

1101 %output folds simply displays text output that has been generated by a parent
1102 fold. It is not executable and therefore it is not highlighted in light blue like
1103 executable folds are.

1104 10.3.5 %error

1105 %error folds display error messages that have been sent by the software that
1106 was executing the code in a fold.

1107 10.3.6 %html

1108 %html folds display HTML code in a floating window as shown in the following
1109 example:

```
1110 1:%html,x_size="700",y_size="440"
1111 2:
1112 3:<html>
1113 4:   <h1 align="center">HTML Color Values</h1>
1114 5:   <table border="0" cellpadding="10" cellspacing="1" width="600">
1115 6:     <tr>
1116 7:       <th bgcolor="white" colspan="2"></th>
1117 8:       <th colspan="6">where blue=cc</th>
1118 9:     </tr>
1119 10:    <tr>
1120 11:      <th rowspan="6">where&nbsp;red=</th>
1121 12:      <th>ff</th>
1122 13:      <th bgcolor="#ff00cc">ff00cc</th>
1123 14:      <th bgcolor="#ff33cc">ff33cc</th>
1124 15:      <th bgcolor="#ff66cc">ff66cc</th>
1125 16:      <th bgcolor="#ff99cc">ff99cc</th>
1126 17:      <th bgcolor="#ffcccc">ffcccc</th>
1127 18:      <th bgcolor="#ffffff">ffffcc</th>
1128 19:    </tr>
1129 20:    <tr>
1130 21:      <th>cc</th>
1131 22:      <th bgcolor="#cc00cc">cc00cc</th>
1132 23:      <th bgcolor="#cc33cc">cc33cc</th>
1133 24:      <th bgcolor="#cc66cc">cc66cc</th>
1134 25:      <th bgcolor="#cc99cc">cc99cc</th>
1135 26:      <th bgcolor="#ccccc">ccccc</th>
1136 27:      <th bgcolor="#ccffcc">ccffcc</th>
1137 28:    </tr>
1138 29:    <tr>
1139 30:      <th>99</th>
1140 31:      <th bgcolor="#9900cc">
1141 32:        <font color="#ffffff">9900cc</font>
```

```

1142 33:         </th>
1143 34:         <th bgcolor="#9933cc">9933cc</th>
1144 35:         <th bgcolor="#9966cc">9966cc</th>
1145 36:         <th bgcolor="#9999cc">9999cc</th>
1146 37:         <th bgcolor="#99cccc">99cccc</th>
1147 38:         <th bgcolor="#99ffcc">99ffcc</th>
1148 39:     </tr>
1149 40:     <tr>
1150 41:         <th>66</th>
1151 42:         <th bgcolor="#6600cc">
1152 43:             <font color="#ffffff">6600cc</font>
1153 44:         </th>
1154 45:         <th bgcolor="#6633cc">
1155 46:             <font color="#FFFFFF">6633cc</font>
1156 47:         </th>
1157 48:         <th bgcolor="#6666cc">6666cc</th>
1158 49:         <th bgcolor="#6699cc">6699cc</th>
1159 50:         <th bgcolor="#66cccc">66cccc</th>
1160 51:         <th bgcolor="#66ffcc">66ffcc</th>
1161 52:     </tr>
1162 53:     <tr>
1163 54:         <th colspan="1"></th>
1164 55:         <th>00</th>
1165 56:         <th>33</th>
1166 57:         <th>66</th>
1167 58:         <th>99</th>
1168 59:         <th>cc</th>
1169 60:         <th>ff</th>
1170 61:     </tr>
1171 62:     <tr>
1172 63:         <th colspan="2"></th>
1173 64:         <th colspan="4">where green=</th>
1174 65:     </tr>
1175 66: </table>
1176 67: </html>
1177 68:
1178 69: %/html
1179 70:
1180 71: %output,preserve="false"
1181 72:
1182 73: %/output
1183 74:

```

1184 This code produces the following output:

HTML Color Values

		where blue=cc					
where red=	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=					

1185 The %html fold's **width** and **height** properties determine the size of the display
 1186 window.

1187 10.3.7 %beanshell

1188 BeanShell (<http://beanshell.org>) is a scripting language that uses Java syntax.
 1189 MathRider uses BeanShell as its primary customization language and %beanshell
 1190 folds give MathRider worksheets full access to the internals of MathRider along
 1191 with the functionality provided by plugins. %beanshell folds are an advanced
 1192 topic that will be covered in later books.

1193 10.4 Automatically Inserting Folds & Removing Unpreserved Folds

1194 Typing the the top and bottom fold lines (for example: %mathpiper ...
 1195 %/mathpiper) can be tedious and MathRider has a way to automatically insert
 1196 them. Place the cursor on a line in a .mrw worksheet file where you would like a
 1197 fold inserted and then **press the right mouse button**. A popup menu will be
 1198 displayed which will allow you to have a fold automatically inserted into the
 1199 worksheet at position of the cursor.

1200 This popup menu also has a menu item called "**Remove Unpreserved Folds**". If
 1201 this menu item is selected, all folds which have a "**preserve="false"**" property
 1202 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

1236 **division** operator, % is the **remainder** operator, and ^ is the **exponent**
1237 operator. MathPiper has more operators in addition to these and some of them
1238 will be covered later.

1239 The following examples show the −, *, /, %, and ^ operators being used:

1240 In> 5 - 2
1241 Result> 3

1242 In> 3*4
1243 Result> 12

1244 In> 30/3
1245 Result> 10

1246 In> 8%5
1247 Result> 3

1248 In> 2^3
1249 Result> 8

1250 The − character can also be used to indicate a negative number:

1251 In> -3
1252 Result> -3

1253 Subtracting a negative number results in a positive number:

1254 In> - -3
1255 Result> 3

1256 In MathPiper, **operators** are symbols (or groups of symbols) which are
1257 implemented with **functions**. One can either call the function an operator
1258 represents directly or use the operator to call the function indirectly. However,
1259 using operators requires less typing and they often make a program easier to
1260 read.

1261 **11.3 Operator Precedence**

1262 When expressions contain more than 1 operator, MathPiper uses a set of rules
1263 called **operator precedence** to determine the order in which the operators are
1264 applied to the values in the expression. Operator precedence is also referred to
1265 as the **order of operations**. Operators with higher precedence are evaluated
1266 before operators with lower precedence. The following table shows a subset of
1267 MathPiper's operator precedence rules with higher precedence operators being
1268 placed higher in the table:

1269 ^ Exponents are evaluated right to left.

```

1270    *,%,/ Then multiplication, remainder, and division operations are evaluated
1271    left to right.

```

1272 +, - Finally, addition and subtraction are evaluated left to right.

1273 Lets manually apply these precedence rules to the multi-operator expression we
1274 used earlier. Here is the expression in source code form:

1275	$5 + 6 \cdot 21/18 - 2^3$
------	---------------------------

1276 And here it is in traditional form:

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

1277 According to the precedence rules, this is the order in which MathPiper
1278 evaluates the operations in this expression:

1279 $5 + 6 * 21 / 18 - 2^3$

1280 5 + 6*21/18 - 8

$$1281 \quad 5 + 126/18 - 8$$

1282 5 + 7 - 8

1283 **12** - 8

1284 **4**

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

1289 **11.4 Changing The Order Of Operations In An Expression**

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

1295 In> 2 + 4*5

```
1296 Result> 22
```

1297 If parentheses are placed around $4 + 5$, however, the addition operator is forced
1298 to be evaluated before the multiplication operator and the result is 30:

1299 In> (2 + 4)*5
1300 Result> 30

1301 Parentheses can also be nested and nested parentheses are evaluated from the
1302 most deeply nested parentheses outward:

1303 In> ((2 + 4)*3)*5
1304 Result> 90

1305 Since parentheses are evaluated before any other operators, they are placed at
1306 the top of the precedence table:

- 1307 () Parentheses are evaluated from the inside out.
- 1308 ^ Then exponents are evaluated right to left.
- 1309 *,%,/ Then multiplication, remainder, and division operations are evaluated
1310 left to right.
- 1311 +, - Finally, addition and subtraction are evaluated left to right.

1312 **11.5 Variables**

1313 As discussed in section [7.1.2.1](#), variables are symbols that can be associated with
1314 values. One way to create variables in MathPiper is through **assignment** and
1315 this consists of placing the name of a variable you would like to create on the left
1316 side of an assignment operator **:=** and an expression on the right side of this
1317 operator. When the expression returns a value, the value is assigned (or **bound**
1318 to) to the variable.

1319 In the following example, a variable called **box** is created and the number **7** is
1320 assigned to it:

1321 In> box := 7
1322 Result> 7

1323 Notice that the assignment operator returns the value that was bound to the
1324 variable as its result. If you want to see the value that the variable box (or any
1325 variable) has been bound to, simply evaluate it:

1326 In> box
1327 Result> 7

1328 If a variable has not been bound to a value yet, it will return itself as the result
1329 when it is evaluated:

```
1330 In> box2
1331 Result> box2
```

1332 MathPiper variables are **case sensitive**. This means that MathPiper takes into
1333 account the **case** of each letter in a variable name when it is deciding if two or
1334 more variable names are the same variable or not. For example, the variable
1335 name **Box** and the variable name **box** are not the same variable because the first
1336 variable name starts with an upper case 'B' and the second variable name starts
1337 with a lower case 'b'.

1338 Programs are able to have more than 1 variable and here is a more sophisticated
1339 example which uses 3 variables:

```
1340 a := 2
1341 Result> 2
```

```
1342 b := 3
1343 Result> 3
```

```
1344 a + b
1345 Result> 5
```

```
1346 answer := a + b
1347 Result> 5
```

```
1348 answer
1349 Result> 5
```

1350 The part of an expression that is on the right side of an assignment operator is
1351 always evaluated first and the result is then assigned to the variable that is on
1352 the left side of the operator.

1353 **11.6 Functions & Function Names**

1354 In programming, **functions** are named blocks of code that can be executed one
1355 or more times by being **called** from other parts of the same program or called
1356 from other programs. Functions can have values passed to them from the calling
1357 code and they always return a value back to the calling code when they are
1358 finished executing. An example of a function is the Even() function which was
1359 discussed in an previous section.

1360 Functions are one way that MathPiper enables code to be reused. Most
1361 programming languages allow code to be reused in this way, although in other
1362 languages these named blocks of code are sometimes called **subroutines**,
1363 **procedures**, **methods**, etc.

1364 The functions that come with MathPiper have names which consist of either a
1365 single word (such as **Even()**) or multiple words that have been put together to

1366 form a compound word (such as **IsBound()**). All letters in the names of
1367 functions which come with MathPiper are lower case except the beginning letter
1368 in each word, which are upper case.

1369 **11.7 Functions That Produce Side Effects**

1370 Most functions are executed to obtain the results they produce but some
1371 functions are executed in order have them perform work that is not in the form
1372 of a result. Functions that perform work that is not in the form of a result are
1373 said to produce **side effects**. Side effects include many forms of work such as
1374 sending information to the user, opening files, and changing values in memory.

1375 When a function produces a side effect which sends information to the user, this
1376 information has the words **Side effects:** placed before it instead of the word
1377 **Result:**. The **Echo()** function is an example of a function that produces a side
1378 effect and it is covered in the following section.

1379 **11.7.1 The Echo() and Write() Functions**

1380 The Echo() and Write() functions both send information to the user and this is
1381 often referred to as "printing" in this document. It may also be called "echoing"
1382 and "writing".

1383 **11.7.1.1 Echo()**

1384 The **Echo()** function takes one expression (or multiple expressions separated by
1385 commas) evaluates each expression, and then prints the results as side effect
1386 output. The following examples illustrate this:

```
1387 In> Echo(1)
1388 Result> True
1389 Side Effects>
1390 1
```

1391 In this example, the number 1 was passed to the Echo() function, the number
1392 was evaluated (all numbers evaluate to themselves), and the result of the
1393 evaluation was then printed as a side effect. Notice that Echo() **also returned a**
1394 **result**. In MathPiper, all functions return a result but functions whose main
1395 purpose is to produce a side effect usually just return a result of **True** if the side
1396 effect succeeded or **False** if it failed. In this case, Echo() returned a result of
1397 **True** because it was able to successfully print a 1 as its side effect.

1398 The next example shows multiple expressions being sent to Echo() (notice that
1399 the expressions are separated by commas):

```
1400 In> Echo(1,1+2,2*3)
1401 Result> True
```

```
1402 Side Effects>
1403 1 3 6
```

1404 The expressions were each evaluated and their results were returned as side
1405 effect output.

1406 Each time an Echo() function is executed, it always forces the display to drop
1407 down to the next line after it is finished. This can be seen in the following
1408 program which is similar to the previous one except it uses a separate Echo()
1409 function to display each expression:

```
1410 1:%mathpiper
1411 2:
1412 3:Echo(1);
1413 4:
1414 5:Echo(1+2);
1415 6:
1416 7:Echo(2*3);
1417 8:
1418 9:%/mathpiper
1419 10:
1420 11:    %output,preserve="false"
1421 12:    Result: True
1422 13:
1423 14:    Side effects:
1424 15:    1
1425 16:    3
1426 17:    6
1427 18:    %/output
```

1428 Notice how the 1, the 3, and the 6 are each on their own line.

1429 Now that we have seen how Echo() works, lets use it to do something useful. If
1430 more than one expression is evaluated in a %mathpiper fold, only the result from
1431 the bottommost expression is displayed:

```
1432 1:%mathpiper
1433 2:
1434 3:a := 1;
1435 4:b := 2;
1436 5:c := 3;
1437 6:
1438 7:%/mathpiper
1439 8:
1440 9:    %output,preserve="false"
1441 10:    Result: 3
1442 11:    %/output
```

1443 In MathPiper, programs are executed one line at a time, starting at the topmost

1444 line of code and working downwards from there. In this example, the line `a := 1;`
1445 is executed first, then the line `b := 2;` is executed, and so on. Notice, however,
1446 that even though we wanted to see what was in all three variables, only the
1447 content of the last variable was displayed.

1448 The following example shows how `Echo()` can be used display the contents of all
1449 three variables:

```
1450 1:%mathpiper
1451 2:
1452 3:a := 1;
1453 4:Echo(a);
1454 5:
1455 6:b := 2;
1456 7:Echo(b);
1457 8:
1458 9:c := 3;
1459 10:Echo(c);
1460 11:
1461 12:%/mathpiper
1462 13:
1463 14:    %output,preserve="false"
1464 15:    Result: True
1465 16:
1466 17:    Side effects:
1467 18:    1
1468 19:    2
1469 20:    3
1470 21:    %/output
```

1471 11.7.1.2 Write()

1472 The **Write()** function is similar to the `Echo()` function except it does not
1473 automatically drop the display down to the next line after it finishes executing:

```
1474 1:%mathpiper
1475 2:
1476 3:Write(1);
1477 4:
1478 5:Write(1+2);
1479 6:
1480 7:Echo(2*3);
1481 8:
1482 9:%/mathpiper
1483 10:
1484 11:    %output,preserve="false"
1485 12:    Result: True
1486 13:
```



```
1487 14:      Side effects:
1488 15:      1 3 6
1489 16:      %/output
```

1490 Write() and Echo() have other differences than the one discussed here and more
1491 information about them can be found in the documentation for these functions.

1492 **11.8 Expressions Are Separated By Semicolons**

1493 In the previous sections, you may have noticed that all of the expressions that
1494 were executed inside of a **%mathpiper** fold had a semicolon (;) after them but
1495 the expressions executed in the **MathPiper console** did not have a semicolon
1496 after them. MathPiper actually requires that all expressions end with a
1497 semicolon, but one does not need to add a semicolon to an expression which is
1498 typed into the MathPiper console because the console adds it automatically when
1499 the expression is executed.

1500 All the previous code examples have had each of their expressions on a separate
1501 line, but multiple expressions can also be placed on a single line because the
1502 semicolons tell MathPiper where one expression ends and the next one begins:

```
1503 1:%mathpiper
1504 2:
1505 3:a := 1; Echo(a); b := 2; Echo(b); c := 3; Echo(c);
1506 4:
1507 5:%/mathpiper
1508 6:
1509 7:      %output,preserve="false"
1510 8:      Result: True
1511 9:
1512 10:     Side effects:
1513 11:     1
1514 12:     2
1515 13:     3
1516 14:     %/output
```

1517 The spaces that are in the code on line 2 of this example are used to make the
1518 code more readable. Any spaces that are present within any expressions or
1519 between them are ignored by MathPiper and if we removed the spaces from the
1520 previous code, the output remains the same:

```
1521 1:%mathpiper
1522 2:
1523 3:a:=1;Echo(a);b:=2;Echo(b);c:= 3;Echo(c);
1524 4:
1525 5:%/mathpiper
1526 6:
1527 7:      %output,preserve="false"
```

```
1528 8:      Result: True
1529 9:
1530 10:     Side effects:
1531 11:      1
1532 12:      2
1533 13:      3
1534 14:     %/output
```

1535 11.9 Strings

1536 A **string** is a **value** that is used to hold text-based information. The typical
1537 expression that is used to create a string consists of **text which is enclosed**
1538 **within double quotes**. Strings can be assigned to variables just like numbers
1539 can and strings can also be displayed using the Echo() function. The following
1540 program assigns a string value to the variable 'a' and then echos it to the user:

```
1541 1:%mathpiper
1542 2:
1543 3:a := "Hello, I am a string.";
1544 4:Echo(a);
1545 5:
1546 6:%/mathpiper
1547 7:
1548 8:     %output,preserve="false"
1549 9:     Result: True
1550 10:
1551 11:     Side effects:
1552 12:     Hello, I am a string.
1553 13:     %/output
```

1554 A useful aspect of using MathPiper inside of MathRider is that variables that are
1555 assigned inside of a **%mathpiper fold** are accessible inside of the **MathPiper**
1556 **console** and variables that are assigned inside of the **MathPiper console** are
1557 available inside of **%mathpiper folds**. For example, after the above fold is
1558 executed, the string that has been bound to variable 'a' can be displayed in the
1559 MathPiper console:

```
1560 In> a
1561 Result> "Hello, I am a string."
```

1562 Individual characters in a string can be accessed by placing the character's
1563 position inside of brackets [] after the variable it is assigned. A character's
1564 position is determined by its distance from the left side of the string, starting at
1565 1. For example, in the above string, 'H' is at position 1, 'e' is at position 2, etc.
1566 The following code shows individual characters in the above string being
1567 accessed:

```
1568 In> a[1]
1569 Result> "H"
```

```
1570 In> a[2]
1571 Result> "e"
```

```
1572 In> a[3]
1573 Result> "l"
```

```
1574 In> a[4]
1575 Result> "l"
```

```
1576 In> a[5]
1577 Result> "o"
```

1578 A range of characters in a string can be accessed by using the .. "range"
1579 operator:

```
1580 In> a[8 .. 11]
1581 Result> "I am"
```

1582 The .. operator is covered in section [11.17.3.1. The .. Range Operator](#).

1583 **11.10 Comments**

1584 Source code can often be difficult to understand and therefore all programming
1585 languages provide the ability for **comments** to be included in the code.

1586 Comments are used to explain what the code near them is doing and they are
1587 usually meant to be read by humans instead of being processed by a computer.
1588 Comments are ignored when the program is executed.

1589 There are two ways that MathPiper allows comments to be added to source code.
1590 The first way is by placing two forward slashes // to the left of any text that is
1591 meant to serve as a comment. The text from the slashes to the end of the line
1592 the slashes are on will be treated as a comment. Here is a program that contains
1593 comments which use slashes:

```
1594 1:%mathpiper
1595 2://This is a comment.
1596 3:
1597 4:x := 2; //Set the variable x equal to 2.
1598 5:
1599 6:
1600 7:%/mathpiper
1601 8:
1602 9:    %output,preserve="false"
1603 10:    Result: 2
```

1604 11: %/output

1605 When this program is executed, any text that starts with slashes is ignored.

1606 The second way to add comments to a MathPiper program is by enclosing the
1607 comments inside of slash-asterisk/asterisk-slash symbols `/* */`. This option is
1608 useful when a comment is too large to fit on one line. Any text between these
1609 symbols is ignored by the computer. This program shows a longer comment
1610 which has been placed between these symbols:

```
1611 1:%mathpiper
1612 2:
1613 3:/*
1614 4: This is a longer comment and it uses
1615 5: more than one line. The following
1616 6: code assigns the number 3 to variable
1617 7: x and then returns it as a result.
1618 8:*/
1619 9:
1620 10:x := 3;
1621 11:
1622 12:%/mathpiper
1623 13:
1624 14: %output,preserve="false"
1625 15:     Result: 3
1626 16: %/output
```

1627 11.11 Conditional Operators

1628 A conditional operator is an operator that is used to compare two values.
1629 Expressions that contain conditional operators return a **boolean value** and a
1630 **boolean value** is one that can either be **True** or **False**. Table 2 shows the
1631 conditional operators that MathPiper uses:

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

Table 2: Conditional Operators

1632 The following examples show each of the conditional operators in Table 2 being
 1633 used to compare values that have been assigned to variables **x** and **y**:

```

1634 1:%mathpiper
1635 2:
1636 2:// Example 1.
1637 3:x := 2;
1638 4:y := 3;
1639 5:
1640 6:Echo(x, "=", y, ":", x = y);
1641 7:Echo(x, "!= ", y, ":", x != y);
1642 8:Echo(x, "< ", y, ":", x < y);
1643 9:Echo(x, "<= ", y, ":", x <= y);
1644 10:Echo(x, "> ", y, ":", x > y);
1645 11:Echo(x, ">= ", y, ":", x >= y);
1646 12:
1647 13:%/mathpiper
1648 14:
1649 15:    %output,preserve="false"
1650 16:    Result: True
1651 17:
1652 18:    Side effects:
1653 19:    2 = 3 :False
1654 20:    2 != 3 :True
1655 21:    2 < 3 :True
1656 22:    2 <= 3 :True
1657 23:    2 > 3 :False
1658 24:    2 >= 3 :False
1659 25:    %/output

```

```
1660 1: %mathpiper
1661 2:
1662 3: // Example 2.
1663 4: x := 2;
1664 5: y := 2;
1665 6:
1666 7: Echo(x, "=", y, ":", x = y);
1667 8: Echo(x, "!= ", y, ":", x != y);
1668 9: Echo(x, "< ", y, ":", x < y);
1669 10: Echo(x, "<=", y, ":", x <= y);
1670 11: Echo(x, "> ", y, ":", x > y);
1671 12: Echo(x, ">=", y, ":", x >= y);
1672 13:
1673 14: %/mathpiper
1674 15:
1675 16: %output,preserve="false"
1676 17: Result: True
1677 18:
1678 19: Side effects:
1679 20: 2 = 2 :True
1680 21: 2 != 2 :False
1681 22: 2 < 2 :False
1682 23: 2 <= 2 :True
1683 24: 2 > 2 :False
1684 25: 2 >= 2 :True
1685 25: %/output
```

```
1686 1: %mathpiper
1687 2:
1688 3: // Example 3.
1689 4: x := 3;
1690 5: y := 2;
1691 6:
1692 7: Echo(x, "=", y, ":", x = y);
1693 8: Echo(x, "!= ", y, ":", x != y);
1694 9: Echo(x, "< ", y, ":", x < y);
1695 10: Echo(x, "<=", y, ":", x <= y);
1696 11: Echo(x, "> ", y, ":", x > y);
1697 12: Echo(x, ">=", y, ":", x >= y);
1698 13:
1699 14: %/mathpiper
1700 15:
1701 16: %output,preserve="false"
1702 17: Result: True
1703 18:
1704 19: Side effects:
1705 20: 3 = 2 :False
1706 21: 3 != 2 :True
```

```
1707 22:      3 < 2 :False
1708 23:      3 <= 2 :False
1709 24:      3 > 2 :True
1710 25:      3 >= 2 :True
1711 26:      %/output
```

1712 Conditional operators are placed at a lower level of precedence than the other
1713 operators we have covered to this point:

1714 () Parentheses are evaluated from the inside out.
1715 ^ Then exponents are evaluated right to left.
1716 *,%,/ Then multiplication, remainder, and division operations are evaluated
1717 left to right.
1718 +, - Then addition and subtraction are evaluated left to right.
1719 =,!=,<,<=,>,>= Finally, conditional operators are evaluated.

1720 ***11.12 Making Decisions With The If() Function & Predicate Expressions***

1721 All programming languages provide the ability to make decisions and the most
1722 commonly used function for making decisions in MathPiper is the **If()** function.
1723 There are two calling formats for the If() function:

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

1724 A **predicate** is an expression which evaluates to either **True** or **False**. The way
1725 the first form of the If() function works is that it evaluates the first expression in
1726 its argument list (which is the "predicate" expression) and then looks at the value
1727 that is returned. If this value is **True**, the "then" expression that is listed second
1728 in the argument list is executed. If the predicate expression evaluates to **False**,
1729 the "then" expression is not executed.

1730 The following program uses an If() function to determine if the number in
1731 variable x is greater than 5. If x is greater than 5, the program will echo
1732 "Greater" and then "End of program":

```
1733 1:%mathpiper
1734 2:
1735 3:x := 6;
1736 4:
1737 5:If(x > 5, Echo(x, "is greater than 5.));
1738 6:
1739 7:Echo("End of program.");
```

```
1740 8:
1741 9: %/mathpiper
1742 10:
1743 11: %output,preserve="false"
1744 12: Result: True
1745 13:
1746 14: Side effects:
1747 15: 6 is greater than 5.
1748 16: End of program.
1749 17: %/output
```

1750 In this program, x has been set to 6 and therefore the expression $x > 5$ is **True**.
1751 When the If() functions evaluates the predicate expression and determines it is
1752 **True**, it then executes the Echo() function. The second Echo() function at the
1753 bottom of the program prints "End of program" regardless of what the If()
1754 function does.

1755 Here is the same program except that x has been set to **4** instead of **6**:

```
1756 1: %mathpiper
1757 2:
1758 3: x := 4;
1759 4:
1760 5: If(x > 5, Echo(x, "is greater than 5.));
1761 6:
1762 7: Echo("End of program.");
1763 8:
1764 9: %/mathpiper
1765 10:
1766 11: %output,preserve="false"
1767 12: Result: True
1768 13:
1769 14: Side effects:
1770 15: End of program.
1771 16: %/output
```

1772 This time the expression $x > 4$ returns a value of **False** which causes the If()
1773 function to not execute the "then" expression that was passed to it.

1774 The second form of the If() function takes a third "else" expression which is
1775 executed only if the predicate expression is **False**. This program is similar to the
1776 previous one except an "else" expression has been added to it:

```
1777 1: %mathpiper
1778 2:
1779 3: x := 4;
1780 4:
1781 5: If(x > 5, Echo(x, "is greater than 5."), Echo(x, "is NOT greater than 5.));
1782 6:
```



```
1783 7:Echo("End of program.");
1784 8:
1785 9:%/mathpiper
1786 10:
1787 11:    %output,preserve="false"
1788 12:    Result: True
1789 13:
1790 14:    Side effects:
1791 15:    4 is NOT greater than 5.
1792 16:    End of program.
1793 17:    %/output
```

1794 11.13 The And(), Or(), & Not() Boolean Functions & Infix Notation

1795 11.13.1 And()

1796 Sometimes one needs to check if two or more expressions are all **True** and one
1797 way to do this is with the **And()** function. The And() function has two calling
1798 formats and this is the first one:

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

1799 This calling format is able to accept one or more expressions as input. If all of
1800 these expressions returns a value of **True**, the And() function will also return a
1801 **True**. However, if any of the expressions returns a **False**, then the And()
1802 function will return a **False**. This can be seen in the following examples:

```
1803 In> And(True, True)
1804 Result> True
```

```
1805 In> And(True, False)
1806 Result> False
```

```
1807 In> And(False, True)
1808 Result> False
```

```
1809 In> And(True, True, True, True)
1810 Result> True
```

```
1811 In> And(True, True, False, True)
1812 Result> False
```

1813 The second format (or **notation**) that can be used to call the And() function is
1814 called **infix** notation:

```
expression1 And expression2
```

1815 With **infix** notation, an expression is placed on both sides of the And() function
1816 name instead of being placed inside of parentheses that are next to it:

```
1817 In> True And True  
1818 Result> True
```

```
1819 In> True And False  
1820 Result> False
```

```
1821 In> False And True  
1822 Result> False
```

1823 Infix notation can only accept two expressions at a time, but it is often more
1824 convenient to use than function calling notation. The following program
1825 demonstrates using the infix version of the And() function:

```
1826 1:%mathpiper  
1827 2:  
1828 3:a := 7;  
1829 4:b := 9;  
1830 5:  
1831 6:Echo("1: ", a < 5 And b < 10);  
1832 7:Echo("2: ", a > 5 And b > 10);  
1833 8:Echo("3: ", a < 5 And b > 10);  
1834 9:Echo("4: ", a > 5 And b < 10);  
1835 10:  
1836 11:If(a > 5 And b < 10, Echo("These expressions are both true.));  
1837 12:  
1838 13:%/mathpiper  
1839 14:  
1840 15:    %output,preserve="false"  
1841 16:    Result: True  
1842 17:  
1843 18:    Side effects:  
1844 19:    1: False  
1845 20:    2: False  
1846 21:    3: False  
1847 22:    4: True  
1848 23:    These expressions are both true.  
1849 23:    %/output
```

1850 11.13.2 Or()

1851 The Or() function is similar to the And() function in that it has both a function

1852 and an infix calling format and it only works with boolean values. However,
1853 instead of requiring that all expressions be **True** in order to return a **True**, Or()
1854 will return a **True** if **one or more expressions are True**.

1855 Here is the function calling format for Or():

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

1856 and these examples show Or() being used with this format:

1857 In> Or(True, False)

1858 Result> True

1859 In> Or(False, True)

1860 Result> True

1861 In> Or(False, False)

1862 Result> False

1863 In> Or(False, False, False, False)

1864 Result> False

1865 In> Or(False, True, False, False)

1866 Result> True

1867 The infix notation format for Or() is as follows:

```
expression1 Or expression2
```

1868 and these examples show this notation being used:

1869 In> True Or False

1870 Result> True

1871 In> False Or True

1872 Result> True

1873 In> False Or False

1874 Result> False

1875 The following program also demonstrates using the infix version of the Or()
1876 function:

1877 1:%mathpiper

1878 2:

1879 3:a := 7;

```
1880 4:b := 9;
1881 5:
1882 6:Echo("1: ", a < 5 Or b < 10);
1883 7:Echo("2: ", a > 5 Or b > 10);
1884 8:Echo("3: ", a > 5 Or b < 10);
1885 9:Echo("4: ", a < 5 Or b > 10);
1886 10:
1887 11:If(a < 5 Or b < 10,Echo("At least one of these expressions is true.));
1888 12:
1889 13:/mathpiper
1890 14:
1891 15:    %output,preserve="false"
1892 16:    Result: True
1893 17:
1894 18:    Side effects:
1895 19:    1: True
1896 20:    2: True
1897 21:    3: True
1898 22:    4: False
1899 23:    At least one of these expressions is true.
1900 24:    %/output
```

1901 11.13.3 Not() & Prefix Notation

1902 The **Not()** function works with boolean expressions like the And() and Or()
1903 functions do, except it can only accept one expression as input. The way Not()
1904 works is that it changes a **True** value to a **False** value and a **False** value to a
1905 **True** value. Here is the Not() function's normal calling format:

```
Not(expression)
```

1906 and these examples show Not() being used with this format:

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

```
1909 In> Not(False)
1910 Result> True
```

1911 Instead of providing an alternative infix calling format like And() and Or() do,
1912 Not()'s second calling format uses **prefix** notation:

```
Not expression
```

1913 Prefix notation looks similar to function notation except no parentheses are used:

```
1914 In> Not True
1915 Result> False
```

```
1916 In> Not False
1917 Result> True
```

1918 Finally, here is a program that uses the prefix version of Not():

```
1919 1:%mathpiper
1920 2:
1921 3:Echo("3 = 3 is ", 3 = 3);
1922 4:
1923 5:Echo("Not 3 = 3 is ", Not 3 = 3);
1924 6:
1925 7:%/mathpiper
1926 8:
1927 9:    %output,preserve="false"
1928 10:    Result: True
1929 11:
1930 12:    Side effects:
1931 13:    3 = 3 is True
1932 14:    Not 3 = 3 is False
1933 15:    %/output
```

1934 **11.14 The While() Looping Function & Bodied Notation**

1935 Many kinds of machines, including computers, derive much of their power from
1936 the principle of **repeated cycling**. **Repeated cycling** in a program means to
1937 execute one or more expressions over and over again and this process is called
1938 "**looping**". MathPiper provides a number of ways to implement loops in a
1939 program and these ways range from straight-forward to subtle.

1940 We will begin discussing looping in MathPiper by starting with the straight-
1941 forward **While** function. The calling format for the **While** function is as follows:

```
1942 While(predicate)
1943 [
1944     body_expressions
1945 ];
```

1946 The **While** function is similar to the **If** function except it will repeatedly execute
1947 the statements it contains as long as its "predicate" expression it **True**. As soon
1948 as the predicate expression returns a **False**, the While() function skips the
1949 expressions it contains and execution continues with the expression that
1950 immediately follows the While() function (if there is one).

1951 The expressions which are contained in a While() function are called its "**body**"

1952 and all functions which have body expressions are called "**bodied**" functions. If
1953 a body contains more than one expression then these expressions need to be
1954 placed within **brackets** []. What body expressions are will become clearer after
1955 looking a some example programs.

1956 The following program uses a While() function to print the integers from 1 to 10:

```
1957 1:%mathpiper
1958 2:
1959 3:// This program prints the integers from 1 to 10.
1960 4:
1961 5:
1962 6:/*
1963 7:    Initialize the variable x to 1
1964 8:    outside of the While "loop".
1965 9:*/
1966 10:x := 1;
1967 11:
1968 12:While(x <= 10)
1969 13:[
1970 14:    Echo(x);
1971 15:
1972 16:    x := x + 1; //Increment x by 1.
1973 17:];
1974 18:
1975 19:%/mathpiper
1976 20:
1977 21:    %output,preserve="false"
1978 22:    Result: True
1979 23:
1980 24:    Side effects:
1981 25:    1
1982 26:    2
1983 27:    3
1984 28:    4
1985 29:    5
1986 30:    6
1987 31:    7
1988 32:    8
1989 33:    9
1990 34:    10
1991 35:    %/output
```

1992 In this program, a single variable called **x** is created. It is used to tell the Echo()
1993 function which **integer** to print and it is also used in the expression that
1994 determines if the While() function should continue to "**loop**" or not.

1995 When the program is executed, 1 is placed into **x** and then the While() function is
1996 called. The predicate expression **x <= 10** becomes **1 <= 10** and, since 1 is less
1997 than or equal to 10, a value of **True** is returned by the expression.

1998 The While() function sees that the expression returned a **True** and therefore it
 1999 executes all of the expressions inside of its **body** from top to bottom.

2000 The Echo() function prints the current contents of x (which is 1) and then the
 2001 expression $x := x + 1$; is executed.

2002 The expression $x := x + 1$; is a standard expression form that is used in many
 2003 programming languages. Each time an expression in this form is evaluated, it
 2004 increases the variable it contains by 1. Another way to describe the effect this
 2005 expression has on x is to say that it **increments** x by **1**.

2006 In this case x contains **1** and, after the expression is evaluated, x contains **2**.

2007 After the last expression inside of a While() function is executed, the While()
 2008 function reevaluates its predicate expression to determine whether it should
 2009 continue looping or not. Since x is **2** at this point, the predicate expression
 2010 returns **True** and the code inside the body of the While() function is executed
 2011 again. This loop will be repeated until x is incremented to **11** and the predicate
 2012 expression returns **False**.

2013 The previous program can be adjusted in a number of ways to achieve different
 2014 results. For example, the following program prints the integers from 1 to 100 by
 2015 changing the **10** in the predicate expression to **100**. A Write() function is used in
 2016 this program so that its output is displayed on the same line until it encounters
 2017 the wrap margin in MathRider (which can be set in Utilities -> Buffer Options...).

```

2018 1:%mathpiper
2019 2:
2020 3:// Print the integers from 1 to 100.
2021 4:
2022 5:x := 1;
2023 6:
2024 7:While(x <= 100)
2025 8:[
2026 9:   Write(x);
2027 10:
2028 11:   x := x + 1; //Increment x by 1.
2029 12:];
2030 13:
2031 14:%/mathpiper
2032 15:
2033 16:   %output,preserve="false"
2034 17:   Result: True
2035 18:
2036 19:   Side effects:
2037 20:   1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
2038      24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
2039      44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
2040      64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83
2041      84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
2042 21:   %/output

```

2043 The following program prints the odd integers from 1 to 99 by changing the
2044 increment value in the increment expression from **1** to **2**:

```
2045 1:%mathpiper
2046 2:
2047 3://Print the odd integers from 1 to 99.
2048 4:
2049 5:x := 1;
2050 6:
2051 7:While(x <= 100)
2052 8:[
2053 9:    Write(x);
2054 10:    x := x + 2; //Increment x by 2.
2055 11:];
2056 12:
2057 13:%/mathpiper
2058 14:
2059 15:    %output,preserve="false"
2060 16:    Result: True
2061 17:
2062 18:    Side effects:
2063 19:    1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43
2064    45 47 49 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83
2065    85 87 89 91 93 95 97 99
2066 20:    %/output
```

2067 Finally, the following program prints the numbers from 1 to 100 in reverse order:

```
2068 1:%mathpiper
2069 2:
2070 3://Print the integers from 1 to 100 in reverse order.
2071 4:
2072 5:x := 100;
2073 6:
2074 7:While(x >= 1)
2075 8:[
2076 9:    Write(x);
2077 10:    x := x - 1; //Decrement x by 1.
2078 11:];
2079 12:
2080 13:%/mathpiper
2081 14:
2082 15:    %output,preserve="false"
2083 16:    Result: True
2084 17:
2085 18:    Side effects:
2086 19:    100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82
2087    81 80 79 78 77 76 75 74 73 72 71 70 69 68 67 66 65 64 63
2088    62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44
```



```

2089         43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25
2090         24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4
2091         3 2 1
2092 20:      %/output

```

2093 In order to achieve the reverse ordering, this program had to initialize **x** to **100**,
 2094 check to see if **x** was **greater than or equal to 1** ($x \geq 1$), and **decrement** **x** by
 2095 **subtracting 1 from it** instead of adding 1 to it.

2096 **11.15 Long-Running Loops, Infinite Loops, & Interrupting Execution**

2097 It is easy to create a loop that will execute a **large number of times**, or even **an**
 2098 **infinite number of times**, either on purpose or by mistake. When you execute
 2099 a program that contains an **infinite loop**, it will run until you tell MathPiper to
 2100 **interrupt** its execution. This is done by selecting the **MathPiper Plugin** (which
 2101 has been placed near the upper left part of the application) and then pressing the
 2102 **"Stop Current Calculation"** button which it contains. (**Note: currently this**
 2103 **button only works if MathPiper is executed inside of a %mathpiper fold.**)

2104 Lets experiment with this button by executing a program that contains an infinite
 2105 loop and then stopping it:

```

2106 1:%mathpiper
2107 2:
2108 3://Infinite loop example program.
2109 4:
2110 5:x := 1;
2111 6:While(x < 10)
2112 7:[
2113 8:    answer := x + 1;
2114 9:];
2115 10:
2116 11:%/mathpiper
2117 12:
2118 13:    %output,preserve="false"
2119 14:    Processing...
2120 15:    %/output

```

2121 Since the contents of **x** is never changed inside the loop, the expression **x < 10**
 2122 always evaluates to **True** which causes the loop to continue looping. Notice that
 2123 the %output fold contains the word **"Processing..."** to indicate that the program
 2124 is executing the code.

2125 Execute this program now and then interrupt it using the **"Stop Current**
 2126 **Calculation"** button. When the program is interrupted, the %output fold will
 2127 display the message **"User interrupted calculation"** to indicate that the
 2128 program was interrupted.

2129 **11.16 Predicate Functions**

2130 A predicate function is a function that either returns **True** or **False**. Most
2131 predicate functions in MathPiper have their names begin with "Is". For example,
2132 IsEven(), IsOdd(), IsInteger, etc. The following examples show some of the
2133 predicate functions that are in MathPiper:

2134 In> IsEven(4)

2135 Result> True

2136 In> IsEven(5)

2137 Result> False

2138 In> IsZero(0)

2139 Result> True

2140 In> IsZero(1)

2141 Result> False

2142 In> IsNegativeInteger(-1)

2143 Result> True

2144 In> IsNegativeInteger(1)

2145 Result> False

2146 In> IsPrime(7)

2147 Result> True

2148 In> IsPrime(100)

2149 Result> False

2150 There is also an IsBound() and an IsUnbound() function that can be used to
2151 determine whether or not a value is bound to a given variable:

2152 In> a

2153 Result> a

2154 In> IsBound(a)

2155 Result> False

2156 In> a := 1

2157 Result> 1

2158 In> IsBound(a)

2159 Result> True

2160 In> Clear(a)

2161 Result> True

```
2162 In> a
2163 Result> a
```

```
2164 In> IsBound(a)
2165 Result> False
```

2166 ***11.17 Lists: Values That Hold Sequences Of Expressions***

2167 The **list** value type is designed to hold expressions in an ordered collection or
2168 sequence. Lists are very flexible and they are one of the most heavily used value
2169 types in MathPiper. Lists can hold expressions of any type, they can grow and
2170 shrink as needed, and they can be nested. Expressions in a list can be accessed
2171 by their position in the list and they can also be replaced by other expressions.

2172 One way to create a list is by placing zero or more objects or expressions inside
2173 of a pair of **braces {}**. The following program creates a list that contains
2174 various expressions and assigns it to the variable x:

```
2175 In> x := {7,42,"Hello",1/2,var}
2176 Result> {7,42,"Hello",1/2,var}
```

```
2177 In> x
2178 Result> {7,42,"Hello",1/2,var}
```

2179 The number of expressions in a list can be determined with the **Length()**
2180 function:

```
2181 In> Length({7,42,"Hello",1/2,var})
2182 Result> 5
```

2183 A single expression in a list can be accessed by placing a set of **brackets []** to
2184 the right of the variable and then putting the expression's position number inside
2185 of the brackets (Notice that the first expression in the list is at position 1
2186 counting from the left side of the list):

```
2187 In> x[1]
2188 Result> 7
```

```
2189 In> x[2]
2190 Result> 42
```

```
2191 In> x[3]
2192 Result> "Hello"
```

```
2193 In> x[4]
2194 Result> 1/2
```

```
2195 In> x[5]
```

2196 `Result> var`

2197 The **1st** and **2nd** expressions in this list are **integers**, the **3rd** expression is a
2198 **string**, the **4th** expression is a **rational number** and the **5th** expression is a
2199 **variable**. Lists can also hold other lists as shown in the following example:

2200 `In> x := {20, 30, {31, 32, 33}, 40}`

2201 `Result> {20,30,{31,32,33},40}`

2202 `In> x[1]`

2203 `Result> 20`

2204 `In> x[2]`

2205 `Result> 30`

2206 `In> x[3]`

2207 `Result> {31,32,33}`

2208 `In> x[4]`

2209 `Result> 40`

2210

2211 The expression in the **3rd** position in the list is another **list** which contains the
2212 expressions **31**, **32**, and **33**. An expression in this second list can be accessed by
2213 two two sets of brackets:

2214 `In> x[3][2]`

2215 `Result> 32`

2216 The **3** inside of the first set of brackets accesses the **3rd** member of the **first** list
2217 and the **2** inside of the second set of brackets accesses the **2nd** member of the
2218 **second** list.

2219 11.17.1 Using While() Loops With Lists

2220 Functions that loop can be used to select each expression in a list in turn so that
2221 an operation can be performed on these expressions. The following program
2222 uses a While() loop to print each of the expressions in a list:

```
2223 1:%mathpiper
2224 2:
2225 3://Print each in in the list.
2226 4:
2227 5:x := {55,93,40,21,7,24,15,14,82};
2228 6:y := 1;
2229 7:
2230 8:While(y <= 9)
2231 9:[
```

```
2232 10:    Echo(y, "- ", x[y]);
2233 11:    y := y + 1;
2234 12:];
2235 13:
2236 14:~/mathpiper
2237 15:
2238 16:    %output,preserve="false"
2239 17:    Result: True
2240 18:
2241 19:    Side effects:
2242 20:    1 - 55
2243 21:    2 - 93
2244 22:    3 - 40
2245 23:    4 - 21
2246 24:    5 - 7
2247 25:    6 - 24
2248 26:    7 - 15
2249 27:    8 - 14
2250 28:    9 - 82
2251 29:~/output
```

2252 A **loop** can also be used to search through a list. The following program uses a
2253 **While()** function and an **If()** function to search through a list to see if it contains
2254 the number **53**. If 53 is found in the list, a message is printed:

```
2255 1:~/mathpiper
2256 2:
2257 3://Determine if 53 is in the list.
2258 4:
2259 5:testList := {18,26,32,42,53,43,54,6,97,41};
2260 6:index := 1;
2261 7:
2262 8:While(index <= 10)
2263 9:[
2264 10:    If(testList[index] = 53,
2265 11:        Echo("53 was found in the list at position", index));
2266 12:
2267 13:    index := index + 1;
2268 14:];
2269 15:
2270 16:~/mathpiper
2271 17:
2272 18:    %output,preserve="false"
2273 19:    Result: True
2274 20:
2275 21:    Side effects:
2276 22:    53 was found in the list at position 5
2277 23:~/output
```

2278 When this program was executed, it determined that **53** was present in the list at
2279 position **5**.

2280 11.17.2 The ForEach() Looping Function

2281 The **ForEach()** function uses a **loop** to index through a list like the While()
2282 function does, but it is more flexible and automatic. ForEach() uses bodied
2283 notation like the While() function does and here is its calling format:

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

2284 **ForEach()** selects each expression in a list in turn, assigns it to the passed-in
2285 "variable", and then executes the expressions that are inside of "body".

2286 Therefore, body is executed once for each expression in the list.

2287 This example shows how ForEach() can be used to print all of the items in a list:

```
2288 1: %mathpiper
2289 2:
2290 3: //Print all values in a list.
2291 4:
2292 5: ForEach(x, {50,51,52,53,54,55,56,57,58,59})
2293 6: [
2294 7:     Echo(x);
2295 8: ];
2296 9:
2297 10: %/mathpiper
2298 11:
2299 12:     %output,preserve="false"
2300 13:     Result: True
2301 14:
2302 15:     Side effects:
2303 16:     50
2304 17:     51
2305 18:     52
2306 19:     53
2307 20:     54
2308 21:     55
2309 22:     56
2310 23:     57
2311 24:     58
2312 25:     59
2313 26: %/output
```

2314 **11.18 Functions & Operators Which Loop Internally To Process Lists**

2315 Looping is such a useful capability that MathPiper has many functions which
2316 loop internally. This section discusses a number of functions that use internal
2317 loops to process lists.

2318 **11.18.1 TableForm()**

```
TableForm(list)
```

2319 The TableForm() function prints the contents of a list in the form of a table. Each
2320 member in the list is printed on its own line and this makes the contents of the
2321 list easier to read:

```
2322 In> testList := {2,4,6,8,10,12,14,16,18,20}
```

```
2323 Result> {2,4,6,8,10,12,14,16,18,20}
```

```
2324 In> TableForm(testList)
```

```
2325 Result> True
```

```
2326 Side Effects>
```

```
2327 2
```

```
2328 4
```

```
2329 6
```

```
2330 8
```

```
2331 10
```

```
2332 12
```

```
2333 14
```

```
2334 16
```

```
2335 18
```

```
2336 20
```

2337 **11.18.2 The .. Range Operator**

```
first .. last
```

2338 One often needs to create a list of consecutive integers and the .. range operator
2339 can be used to do this. The first integer in the list is placed before the ..
2340 operator (with a space in between them) and the last integer in the list is placed
2341 after the .. operator. Here are some examples:

```
2342 In> 1 .. 10
```

```
2343 Result> {1,2,3,4,5,6,7,8,9,10}
```

```
2344 In> 10 .. 1
```

```
2345 Result> {10,9,8,7,6,5,4,3,2,1}
```

```
2346 In> -10 .. 10
2347 Result> {-10,-9,-8,-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,8,9,10}
```

2348 As the examples show, the `..` operator can generate lists of integers in ascending
2349 order and descending order. It can also generate lists that contain negative
2350 integers.

2351 11.18.3 Contains()

2352 The **Contains()** function searches a list to determine if it contains a given
2353 expression. If it finds the expression, it returns **True** and if it doesn't find the
2354 expression, it returns **False**. Here is the calling format for Contains():

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

2355 The following code shows Contains() being used to locate a number in a list:

```
2356 In> Contains({50,51,52,53,54,55,56,57,58,59}, 53)
2357 Result> True
```

```
2358 In> Contains({50,51,52,53,54,55,56,57,58,59}, 75)
2359 Result> False
```

2360 The **Not()** function can also be used with predicate functions like Contains() to
2361 change their results:

```
2362 In> Not Contains({50,51,52,53,54,55,56,57,58,59}, 75)
2363 Result> True
```

2364 11.18.4 Find()

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

2365 The **Find()** function searches a list for the first occurrence of a given expression.
2366 If the expression is found, the numerical position of its first occurrence is
2367 returned and if it is not found, -1 is returned:

```
2368 In> Find({23, 15, 67, 98, 64}, 15)
2369 Result> 2
```

```
2370 In> Find({23, 15, 67, 98, 64}, 8)
2371 Result> -1
```


2372 **11.18.5 Count()**

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

2373 **Count()** determines the number of times a given expression occurs in a list:

2374 In> testList := {a,b,b,c,c,c,d,d,d,d,e,e,e,e,e}

2375 Result> {a,b,b,c,c,c,d,d,d,d,e,e,e,e,e}

2376 In> Count(testList, c)

2377 Result> 3

2378 In> Count(testList, e)

2379 Result> 5

2380 In> Count(testList, z)

2381 Result> 0

2382 **11.18.6 Select()**

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

2383 **Select()** returns a list that contains all the expressions in a list which make a
2384 given predicate return **True**:

2385 In> Select("IsPositiveInteger",{46,87,59,-27,11,86,-21,-58,-86,-52})

2386 Result> {46,87,59,11,86}

2387 In this example, notice that the **name** of the predicate function is passed to
2388 Select() in **double quotes**. There are other ways to pass a predicate function to
2389 Select() but these are covered in a later section.

2390 Here are some further examples which use the Select() function:

2391 In> Select("IsOdd", {16,14,82,92,33,74,99,67,65,52})

2392 Result> {33,99,67,65}

2393 In> Select("IsEven", {16,14,82,92,33,74,99,67,65,52})

2394 Result> {16,14,82,92,74,52}

2395 In> Select("IsPrime", 1 .. 75)

2396 Result> {2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73}

2397 Notice how the third example uses the .. operator to automatically generate a list
2398 of consecutive integers from 1 to 75 for the Select() function to analyze.

2399 11.18.7 The Nth() Function & The [] Operator

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

2400 The **Nth()** function simply returns the expression which is at a given index in a
2401 list. This example shows the third expression in a list being obtained:

```
2402 In> testList := {a,b,c,d,e,f,g}
```

```
2403 Result> {a,b,c,d,e,f,g}
```

```
2404 In> Nth(testList, 3)
```

```
2405 Result> c
```

2406 As discussed earlier, the **[]** operator can also be used to obtain a single
2407 expression from a list:

```
2408 In> testList[3]
```

```
2409 Result> c
```

2410 The **[]** operator can even obtain a single expression directly from a list without
2411 needing to use a variable:

```
2412 In> {a,b,c,d,e,f,g}[3]
```

```
2413 Result> c
```

2414 11.18.8 Append() & Nondestructive List Operations

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

2415 The **Append()** function adds an expression to the end of a list:

```
2416 In> testList := {21,22,23}
```

```
2417 Result> {21,22,23}
```

```
2418 In> Append(testList, 24)
```

```
2419 Result> {21,22,23,24}
```

2420 However, instead of changing the **original** list, MathPiper creates a **copy** of the
2421 **original** list and appends the expression to the **copy**. This can be confirmed by
2422 evaluating the variable **testList** after the Append() function has been called:

```
2423 In> testList
```

```
2424 Result> {21,22,23}
```

2425 Notice that the list that is bound to **testList** was not modified by the Append()
2426 function. This is called a **nondestructive list operation** and most MathPiper
2427 functions that manipulate lists do so nondestructively. To have the changed list
2428 bound to the variable that it being used, the following technique can be
2429 employed:

```
2430 In> testList := {21,22,23}  
2431 Result> {21,22,23}
```

```
2432 In> testList := Append(testList, 24)  
2433 Result> {21,22,23,24}
```

```
2434 In> testList  
2435 Result> {21,22,23,24}
```

2436 After this code has been executed, the modified list has indeed been bound to
2437 testList as desired.

2438 There are some functions, such as DestructiveAppend(), which **do** change the
2439 original list and most of them begin with the word "Destructive". These are
2440 called "destructive functions" and it is recommended that destructive functions
2441 should be used with care.

2442 11.18.9 The : Prepend Operator

```
expression : list
```

2443 The prepend operator is a colon : and it can be used to add an expression to the
2444 beginning of a list:

```
2445 In> testList := {b,c,d}  
2446 Result> {b,c,d}
```

```
2447 In> testList := a:testList  
2448 Result> {a,b,c,d}
```

2449 11.18.10 Concat()

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

2450 The Concat() function is short for "concatenate" which means to join together
2451 sequentially. It takes two or more lists and joins them together into a
2452 single larger list:

```
2453 In> Concat({a,b,c}, {1,2,3}, {x,y,z})
2454 Result> {a,b,c,1,2,3,x,y,z}
```

2455 11.18.11 Insert(), Delete(), & Replace()

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

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

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

2456 **Insert()** inserts an expression into a list at a given index, **Delete()** deletes an
2457 expression from a list at a given index, and **Replace()** replaces an expression in
2458 a list at a given index with another expression:

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

2461 In> testList := Insert(testList, 4, 123)
2462 Result> {a,b,c,123,d,e,f,g}
```

```
2463 In> testList := Delete(testList, 4)
2464 Result> {a,b,c,d,e,f,g}
```

```
2465 In> testList := Replace(testList, 4, xxx)
2466 Result> {a,b,c,xxx,e,f,g}
```

2467 11.18.12 Take()

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

2468 **Take()** obtains a sublist from the **beginning** of a list, the **end** of a list, or the
2469 **middle** of a list. The expressions in the list that are not taken are discarded.

2470 A **positive** integer passed to Take() indicates how many expressions should be
2471 taken from the **beginning** of a list:

2472 In> testList := {a,b,c,d,e,f,g}

2473 Result> {a,b,c,d,e,f,g}

2474 In> Take(testList, 3)

2475 Result> {a,b,c}

2476 A **negative** integer passed to Take() indicates how many expressions should be
2477 taken from the **end** of a list:

2478 In> Take(testList, -3)

2479 Result> {e,f,g}

2480 Finally, if a **two member list** is passed to Take() it indicates the **range** of
2481 expressions that should be taken from the **middle** of a list. The **first** value in the
2482 passed-in list specifies the **beginning** index of the range and the **second** value
2483 specifies its **end**:

2484 In> Take(testList, {3,5})

2485 Result> {c,d,e}

2486 11.18.13 Drop()

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

2487 **Drop()** does the opposite of Take() in that it **drops** expressions from the
2488 **beginning** of a list, the **end** of a list, or the **middle** of a list and **returns a list**
2489 **which contains the remaining expressions.**

2490 A **positive** integer passed to Drop() indicates how many expressions should be
2491 dropped from the **beginning** of a list:

2492 In> testList := {a,b,c,d,e,f,g}

2493 Result> {a,b,c,d,e,f,g}

2494 In> Drop(testList, 3)

2495 Result> {d,e,f,g}

2496 A **negative** integer passed to Drop() indicates how many expressions should be
2497 dropped from the **end** of a list:

2498 In> Drop(testList, -3)

2499 Result> {a,b,c,d}

2500 Finally, if a **two member list** is passed to Drop() it indicates the **range** of
2501 expressions that should be dropped from the **middle** of a list. The **first** value in
2502 the passed-in list specifies the **beginning** index of the range and the **second**
2503 value specifies its **end**:

```
2504 In> Drop(testList, {3,5})  
2505 Result> {a,b,f,g}
```

2506 **11.18.14 FillList()**

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

2507 The FillList() function simply creates a list which is of size "length" and fills it
2508 with "length" copies of the given expression:

```
2509 In> FillList(a, 5)  
2510 Result> {a,a,a,a,a}  
  
2511 In> FillList(42,8)  
2512 Result> {42,42,42,42,42,42,42,42}
```

2513 **11.18.15 RemoveDuplicates()**

```
RemoveDuplicates(list)
```

2514 **RemoveDuplicates()** removes any duplicate expressions that are contained in
2515 in a list:

```
2516 In> testList := {a,a,b,c,c,b,b,a,b,c,c}  
2517 Result> {a,a,b,c,c,b,b,a,b,c,c}  
  
2518 In> RemoveDuplicates(testList)  
2519 Result> {a,b,c}
```

2520 **11.18.16 Reverse()**

```
Reverse(list)
```

2521 **Reverse()** reverses the order of the expressions in a list:

2522 In> testList := {a,b,c,d,e,f,g,h}

2523 Result> {a,b,c,d,e,f,g,h}

2524 In> Reverse(testList)

2525 Result> {h,g,f,e,d,c,b,a}

2526 11.18.17 Partition()

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

2527 The **Partition()** function breaks a list into sublists of size "partition_size":

2528 In> testList := {a,b,c,d,e,f,g,h}

2529 Result> {a,b,c,d,e,f,g,h}

2530 In> Partition(testList, 2)

2531 Result> {{a,b},{c,d},{e,f},{g,h}}

2532 If the partition_size does not divide the length of the list evenly, the remaining
2533 elements are discarded:

2534 In> Partition(testList, 3)

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

2536 The number of elements that Partition() will discard can be calculated by
2537 dividing the length of a list by the partition size and obtaining the remainder:

2538 In> Mod(Length(testList), 3)

2539 Result> 2

2540 The Mod() function, which divides two integers and return their remainder, is
2541 covered in a later section.

2542 11.19 Functions That Work With Integers

2543 This section discusses various functions which work with integers. Some of
2544 these functions also work with non-integer values and their use with non-
2545 integers is discussed in other sections.

2546 11.19.1 RandomIntegerVector()

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

2547 A vector can be thought of as a list that does not contain other lists.
2548 **RandomIntegerVector()** creates a list of size "length" that contains random
2549 integers that are no lower than "lowest_possible" and no higher than "highest
2550 possible". The following example creates **10** random integers between **1** and **99**
2551 inclusive:

```
2552 In> RandomIntegerVector(10, 1, 99)
2553 Result> {73,93,80,37,55,93,40,21,7,24}
```

2554 11.19.2 Max() & Min()

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

2555 If two values are passed to Max(), it determines which one is larger:

```
2556 In> Max(10, 20)
2557 Result> 20
```

2558 If a list of values are passed to Max(), it finds the largest value in the list:

```
2559 In> testList := RandomIntegerVector(10, 1, 99)
2560 Result> {73,93,80,37,55,93,40,21,7,24}
```

```
2561 In> Max(testList)
2562 Result> 93
```

2563 The **Min()** function is the opposite of the Max() function.

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

2564 If two values are passed to Min(), it determines which one is smaller:

```
2565 In> Min(10, 20)
2566 Result> 10
```

2567 If a list of values are passed to Min(), it finds the smallest value in the list:

```
2568 In> testList := RandomIntegerVector(10, 1, 99)
2569 Result> {73,93,80,37,55,93,40,21,7,24}
```

```
2570 In> Min(testList)
2571 Result> 7
```


2572 **11.19.3 Div() & Mod()**

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

2573 **Div()** stands for "divide" and determines the whole number of times a divisor
2574 goes into a dividend:

```
2575 In> Div(7, 3)
2576 Result> 2
```

2577 **Mod()** stands for "modulo" and it determines the remainder that results when a
2578 dividend is divided by a divisor:

```
2579 In> Mod(7,3)
2580 Result> 1
```

2581 The remainder/modulo operator % can also be used to calculate a remainder:

```
2582 In> 7 % 2
2583 Result> 1
```

2584 **11.19.4 Gcd()**

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

2585 GCD stands for Greatest Common Denominator and the **Gcd()** function
2586 determines the greatest common denominator of the values that are passed to it.

2587 If two integers are passed to Gcd(), it calculates their greatest common
2588 denominator:

```
2589 In> Gcd(21, 56)
2590 Result> 7
```

2591 If a list of integers are passed to Gcd(), it finds the greatest common
2592 denominator of all the integers in the list:

```
2593 In> Gcd({9, 66, 123})
2594 Result> 3
```

2595 **11.19.5 Lcm()**

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

2596 LCM stands for Least Common Multiple and the **Lcm()** function determines the
2597 least common multiple of the values that are passed to it.

2598 If two integers are passed to Lcm(), it calculates their least common multiple:

```
2599 In> Lcm(14, 8)
2600 Result> 56
```

2601 If a list of integers are passed to Lcm(), it finds the least common multiple of all
2602 the integers in the list:

```
2603 In> Lcm({3,7,9,11})
2604 Result> 693
```

2605 **11.19.6 Add()**

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

2606 **Add()** can find the sum of two or values passed to it:

```
2607 In> Add(3,8,20,11)
2608 Result> 42
```

2609 It can also find the sum of a list of values :

```
2610 In> testList := RandomIntegerVector(10,1,99)
2611 Result> {73,93,80,37,55,93,40,21,7,24}
```

```
2612 In> Add(testList)
2613 Result> 523
```

```
2614 In> testList := 1 .. 10
2615 Result> {1,2,3,4,5,6,7,8,9,10}
```

```
2616 In> Add(testList)
2617 Result> 55
```

2618 **11.19.7 Factorize()**

```
Factorize(list)
```

2619 This function has two calling formats, only one of which is discussed here.

2620 **Factorize(list)** multiplies all the expressions in a list together and returns their
2621 product:

```
2622 In> Factorize({1,2,3})
```

```
2623 Result> 6
```

2624 **11.20 User Defined Functions**

2625 In computer programming, a **function** is a named sections of code that can be
2626 **called** from other sections of code. **Values** can be sent to a function for
2627 processing as part of the **call** and a function always returns a value as its result.

2628 The values that are sent to a function when it is called are called **arguments** and
2629 a function can accept 0 or more of them. These arguments are placed within
2630 parentheses.

2631 MathPiper has many predefined functions (some of which have been discussed in
2632 previous sections) but users can create their own functions too. The following
2633 program creates a function called **addNums()** which takes two numbers as
2634 arguments, adds them together, and returns their sum back to the calling code
2635 as a result:

```
2636 In> addNums(num1,num2) := num1 + num2
```

```
2637 Result> True
```

2638 This line of code defined a new function called **addNums** and specified that it
2639 will accept two values when it is called. The **first** value will be placed into the
2640 variable **num1** and the **second** value will be placed into the variable **num2**. The
2641 code on the **right side** of the assignment operator is then bound to this function
2642 and it is executed each time the function is called. The following example shows
2643 the new addNums() function being called multiple times with different values
2644 being passed to it:

```
2645 In> addNums(2,3)
```

```
2646 Result> 5
```

```
2647 In> addNums(4,5)
```

```
2648 Result> 9
```

```
2649 In> addNums(9,1)
```

2650 `Result> 10`

2651 Notice that, unlike the functions that come with MathPiper, we chose to have this
2652 function's name start with a **lower case letter**. We could have had addNums()
2653 begin with an upper case letter but it is a convention in MathPiper for user
2654 defined function names to begin with a lower case letter to distinguish them
2655 from the functions that come with MathPiper.

2656 The values that are returned from user defined functions can also be assigned to
2657 variables. The following example uses a %mathpiper fold to define a function
2658 called **evenIntegers()** and then this function is used in the MathPiper console:

```
2659 1:%mathpiper
2660 2:
2661 3:evenIntegers(endInteger) :=
2662 4:[
2663 5:    resultList := {};
2664 6:    x := 2;
2665 7:
2666 8:    While(x <= endInteger)
2667 9:    [
2668 10:        resultList := Append(resultList, x);
2669 11:        x := x + 2;
2670 12:    ];
2671 13:
2672 14:    resultList;
2673 15:];
2674 16:
2675 17:%/mathpiper
2676 18:
2677 19:    %output,preserve="false"
2678 20:    Result: True
2679 21:    %/output
```

2680 `In> a := evenIntegers(10)`

2681 `Result> {2,4,6,8,10}`

2682 `In> Length(a)`

2683 `Result> 5`

2684 The function evenIntegers() returns a list which contains all the even integers
2685 from 2 up through the value that was passed into it. The fold was first executed
2686 in order to define the evenIntegers() function and make it ready for use. The
2687 evenIntegers() function was then called from the MathPiper console and 10 was
2688 passed to it. After the function was finished executing, it return a list of even
2689 integers as a result and this result was assigned to the variable 'a'. We then
2690 passed the list that was assigned to 'a' to the Length() function in order to
2691 determine its size.

2692 **11.20.1 Global Variables, Local Variables, & Local()**

2693 The new evenIntegers() function seems to work well, but there is a problem. The
2694 variables 'x' and resultList were defined inside the function as **global variables**
2695 which means they are accessible from anywhere, including from within other
2696 functions, within folds:

```
2697 1:%mathpiper
2698 2:
2699 3:Echo(x, ",", resultList);
2700 4:
2701 5:%/mathpiper
2702 6:
2703 7:    %output,preserve="false"
2704 8:    Result: True
2705 9:
2706 10:    Side effects:
2707 11:    12 ,{2,4,6,8,10}
2708 12:    %/output
```

2709 and from within the MathPiper console:

```
2710 In> x
2711 Result> 12
2712 In> resultList
2713 Result> {2,4,6,8,10}
```

2714 Using global variables inside of functions is usually not a good idea because code
2715 in other functions and folds might already be using (or will use) the same
2716 variable names. Global variables which have the same name are the same
2717 variable. When one section of code changes the value of a given global variable,
2718 the value is changed everywhere that variable is used and this will eventually
2719 cause errors.

2720 In order to prevent errors like this, a function named **Local()** can be called
2721 inside a function to define what are called **local variables**. A **local variable** is
2722 only accessible inside the function it has been defined in, even if it has the same
2723 name as a global variable. The following example shows a second version of the
2724 evenIntegers() function which uses **Local()** to make **x** and **resultList** local
2725 variables:

```
2726 1:%mathpiper
2727 2:
2728 3:/*
2729 4: This version of evenIntegers() uses Local() to make
2730 5: x and resultList local variables
```

```
2731 6:*/
2732 7:
2733 8:evenIntegers(endInteger) :=
2734 9:[
2735 10:    Local(x,resultList);
2736 11:
2737 12:    resultList := {};
2738 13:    x := 2;
2739 14:
2740 15:    While(x <= endInteger)
2741 16:    [
2742 17:        resultList := Append(resultList, x);
2743 18:        x := x + 2;
2744 19:    ];
2745 20:
2746 21:    resultList;
2747 22:];
2748 23:
2749 24:*/mathpiper
2750 25:
2751 26:    %output,preserve="false"
2752 27:    Result: True
2753 28:    %/output
```

2754 We can verify that x and resultList are now local variables by first clearing them,
2755 calling evenIntegers(), and then seeing what x and resultList contain:

```
2756 In> Clear(x, resultList)
2757 Result> True
```

```
2758 In> evenIntegers(10)
2759 Result> {2,4,6,8,10}
```

```
2760 In> x
2761 Result> x
```

```
2762 In> resultList
2763 Result> resultList
```

2764 11.21 Applying Functions To List Members

2765 11.21.1 Table()

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

2766 The Table() function creates a list of values by doing the following:

- 2767 1) Generating a sequence of values between a "begin_value" and an
2768 "end_value" with each value being incremented by the "step_amount".
- 2769 2) Placing each value in the sequence into the specified "variable", one value
2770 at a time.
- 2771 3) Evaluating the defined "expression" (which contains the defined "variable")
2772 for each value, one at a time.
- 2773 4) Placing the result of each "expression" evaluation into the result list.

2774 This example generates a list which contains the integers 1 through 10:

```
2775 In> Table(x, x, 1, 10, 1)
2776 Result> {1,2,3,4,5,6,7,8,9,10}
```

2777 Notice that the expression in this example is simply the variable itself with no
2778 other operations performed on it.

2779 The following example is similar to the previous one except that its expression
2780 multiplies x by 2:

```
2781 In> Table(x*2, x, 1, 10, 1)
2782 Result> {2,4,6,8,10,12,14,16,18,20}
```

2783 Lists which contain decimal values can also be created by setting the
2784 "step_amount" to a decimal:

```
2785 In> Table(x, x, 0, 1, .1)
2786 Result> {0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1}
```

2787 **12 THE CONTENT BELOW THIS LINE IS STILL UNDER**
2788 **DEVELOPMENT**

2789 **12.1 Sets**

2790 The following example shows operations that MathPiper can perform on sets:

```
2791 a = Set([0,1,2,3,4])
2792 b = Set([5,6,7,8,9,0])
2793 a,b
2794 |
2795 ({0, 1, 2, 3, 4}, {0, 5, 6, 7, 8, 9})
```

```
2796 a.cardinality()
2797 |
2798 5
```

```
2799 3 in a
2800 |
2801 True
```

```
2802 3 in b
2803 |
2804 False
```

```
2805 a.union(b)
2806 |
2807 {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

```
2808 a.intersection(b)
2809 |
2810 {0}
```


2811 **13 Miscellaneous Topics**

2812 **13.1 Errors**

2813 **13.2 Style Guide For Expressions**

2814 Always surround the following binary operators with a single space on either
2815 side: assignment ':=', comparisons (==, <, >, !=, <>, <=, >=, Booleans (and, or,
2816 not).

2817 Use spaces around the + and – arithmetic operators and no spaces around the
2818 *, /, %, and ^ arithmetic operators:

2819 $x = x + 1$

2820 $x = x*3 - 5\%2$

2821 $c = (a + b)/(a - b)$

2822 **13.3 Built-in Constants**

2823 MathPiper has a number of mathematical constants built into it and the following
2824 is a list of some of the more common ones:

2825 Pi, pi: The ratio of the circumference to the diameter of a circle.

2826 E, e: Base of the natural logarithm.

2827 I, i: The imaginary unit quantity.

2828

2829 log2: The natural logarithm of the real number 2.

2830 Infinity, infinity: Can have + or – placed before it to indicate positive or negative
2831 infinity.

2832 **14 Solving Equations**

2833 **14.1 Solving Equations Symbolically**

2834 **14.1.1 Symbolic Expressions & Simplify()**

2835 Expressions that contain symbolic variables are called symbolic expressions. In
2836 the following example, b is defined to be a symbolic variable and then it is used
2837 to create the symbolic expression $2*b$:

```
2838 var('b')
2839 type(2*b)
2840 |
2841 <class 'sage.calculus.calculus.SymbolicArithmetic'>
```

2842 As can be seen by this example, the symbolic expression $2*b$ was placed into an
2843 object of type SymbolicArithmetic. The expression can also be assigned to a
2844 variable:

```
2845 m = 2*b
2846 type(m)
2847 |
2848 <class 'sage.calculus.calculus.SymbolicArithmetic'>
```

2849 The following program creates two symbolic expressions, assigns them to
2850 variables, and then performs operations on them:

```
2851 m = 2*b
2852 n = 3*b
2853 m+n, m-n, m*n, m/n
2854 |
2855 (5*b, -b, 6*b^2, 2/3)
```

2856 Here is another example that multiplies two symbolic expressions together:

```
2857 m = 5 + b
2858 n = 8 + b
2859 y = m*n
2860 y
2861 |
2862 (b + 5)*(b + 8)
```

2863 **14.1.1.1 Expanding And Factoring**

2864 If the expanded form of the expression from the previous section is needed, it is
2865 easily obtained by calling the `expand()` method (this example assumes the cells in
2866 the previous section have been run):

```
2867 z = y.expand()
2868 z
2869 |
2870  $b^2 + 13b + 40$ 
2871 The expanded form of the expression has been assigned to variable z and the
2872 factored form can be obtained from z by using the factor() method:
```

```
2873 z.factor()
2874 |
2875  $(b + 5)(b + 8)$ 
2876 By the way, a number can be factored without being assigned to a variable by
2877 placing parentheses around it and calling its factor() method:
```

```
2878 (90).factor()
2879 |
2880  $2 * 3^2 * 5$ 
```

2881 ***14.1.1.2 Miscellaneous Symbolic Expression Examples***

```
2882 var('a,b,c')
2883  $(5a + b + 4c) + (2a + 3b + c)$ 
2884 |
2885  $5c + 4b + 7a$ 
```

```
2886  $(a + b) - (x + 2b)$ 
2887 |
2888  $-x - b + a$ 
```

```
2889  $3a^2 - a(a - 5)$ 
2890 |
2891  $3a^2 - (a - 5)a$ 
```

```
2892 _.factor()
2893 |
2894  $a(2a + 5)$ 
```

2895 **14.1.2 Symbolic Equations and The solve() Function**

```
2896 In addition to working with symbolic expressions, MathPiper is also able to work
2897 with symbolic equations:
```

```
2898 var('a')
2899  $\text{type}(x^2 == 16a^2)$ 
2900 |
```

```
2901 <class 'sage.calculus.equations.SymbolicEquation'>
2902 As can be seen by this example, the symbolic equation  $x^2 == 16a^2$  was
2903 placed into an object of type SymbolicEquation. A symbolic equation needs to
2904 use double equals '==' so that it can be assigned to a variable using a single
2905 equals '=' like this:

2906 m = x^2 == 16*a^2
2907 m, type(m)
2908 |
2909 (x^2 == 16*a^2, <class 'sage.calculus.equations.SymbolicEquation'>)
2910 Many symbolic equations can be solved algebraically using the solve() function:

2911 solve(m, a)
2912 |
2913 [a == -x/4, a == x/4]
2914 The first parameter in the solve() function accepts a symbolic equation and the
2915 second parameter accepts the symbolic variable to be solved for.

2916 The solve() function can also solve simultaneous equations:

2917 var('i1,i2,i3,v0')

2918 a = (i1 - i3)*2 + (i1 - i2)*5 + 10 - 25 == 0
2919 b = (i2 - i3)*3 + i2*1 - 10 + (i2 - i1)*5 == 0
2920 c = i3*14 + (i3 - i2)*3 + (i3 - i1)*2 - (-3*v0) == 0
2921 d = v0 == (i2 - i3)*3

2922 solve([a,b,c,d], i1,i2,i3,v0)
2923 |
2924 [[i1 == 4, i2 == 3, i3 == -1, v0 == 12]]
2925 Notice that, when more than one equation is passed to solve(), they need to be
2926 placed into a list.
```

2927 **14.2 Solving Equations Numerically**

2928 **14.2.1 Roots**

2929 The sqrt() function can be used to obtain the square root of a value, but a more
2930 general technique is used to obtain other roots of a value. For example, if one
2931 wanted to obtain the cube root of 8:

```
2932 8 would be raised to the 1/3 power:
2933 8^(1/3)
2934 |
2935 2
```

2936 Due to the order of operations, the rational number 1/3 needs to be placed within
2937 parentheses in order for it to be evaluated as an exponent.

2938 ***14.3 Finding Roots Graphically And Numerically With The find_root()***
2939 ***Method***

2940 Sometimes equations cannot be solved algebraically and the solve() function
2941 indicates this by returning a copy of the input it was passed. This is shown in the
2942 following example:

```
2943 f(x) = sin(x) - x - pi/2
2944 eqn = (f == 0)
2945 solve(eqn, x)
2946 |
2947 [x == (2*sin(x) - pi)/2]
```

2948 However, equations that cannot be solved algebraically can be solved both
2949 graphically and numerically. The following example shows the above equation
2950 being solved graphically:

```
2951 show(plot(f,-10,10))
2952 |
```

2953 This graph indicates that the root for this equation is a little greater than -2.5.

2954 The following example shows the equation being solved more precisely using the
2955 find_root() method:

```
2956 f.find_root(-10,10)
2957 |
2958 -2.309881460010057
```

2959 The -10 and +10 that are passed to the find_root() method tell it the interval
2960 within which it should look for roots.

2961 15 Output Forms

2962 15.1 LaTeX Is Used To Display Objects In Traditional Mathematics Form

2963 LaTeX (pronounced lā-tek, <http://en.wikipedia.org/wiki/LaTeX>) is a document
2964 markup language which is able to work with a wide range of mathematical
2965 symbols. MathPiper objects will provide LaTeX descriptions of themselves when
2966 their latex() methods are called. The LaTeX description of an object can also be
2967 obtained by passing it to the latex() function:

```
2968 a = (2*x^2)/7
```

```
2969 latex(a)
```

```
2970 |
```

```
2971 \frac{{2 \cdot {x}^{{2}} }}{{7}}
```

2972 When this result is fed into LaTeX display software, it will generate traditional
2973 mathematics form output similar to the following:

2974 The jsMath package which is referenced in is the software that the MathPiper
2975 Notebook uses to translate LaTeX input into traditional mathematics form
2976 output.

2977 15.2 Displaying Mathematical Objects In Traditional Form

2978 Earlier it was indicated that MathPiper is able to display mathematical objects in
2979 either text form or traditional form. Up until this point, we have been using text
2980 form which is the default. If one wants to display a mathematical object in
2981 traditional form, the show() function can be used. The following example creates
2982 a mathematical expression and then displays it in both text form and traditional
2983 form:

```
2984 var('y,b,c')
```

```
2985 z = (3*y^(2*b))/(4*x^c)^2
```

```
2986 #Display the expression in text form.
```

```
2987 z
```

```
2988 |
```

```
2989 3*y^(2*b)/(16*x^(2*c))
```

```
2990 #Display the expression in traditional form.
```

```
2991 show(z)
```

```
2992 |
```

2993 **16 2D Plotting**

2994 **17 High School Math Problems (most of the problems are still in**
2995 **development)**

2996 **17.1 Pre-Algebra**

2997 Wikipedia entry.

2998 <http://en.wikipedia.org/wiki/Pre-algebra>

2999 (In development...)

3000 **17.1.1 Equations**

3001 Wikipedia entry.

3002 <http://en.wikipedia.org/wiki/Equation>

3003 (In development...)

3004 **17.1.2 Expressions**

3005 Wikipedia entry.

3006 http://en.wikipedia.org/wiki/Mathematical_expression

3007 (In development...)

3008 **17.1.3 Geometry**

3009 Wikipedia entry.

3010 <http://en.wikipedia.org/wiki/Geometry>

3011 (In development...)

3012 **17.1.4 Inequalities**

3013 Wikipedia entry.

3014 <http://en.wikipedia.org/wiki/Inequality>

3015 (In development...)

3016 **17.1.5 Linear Functions**

3017 Wikipedia entry.

3018 http://en.wikipedia.org/wiki/Linear_functions

3019 (In development...)

3020 **17.1.6 Measurement**

3021 Wikipedia entry.

3022 <http://en.wikipedia.org/wiki/Measurement>

3023 (In development...)

3024 17.1.7 Nonlinear Functions

3025 Wikipedia entry.

3026 http://en.wikipedia.org/wiki/Nonlinear_system

3027 (In development...)

3028 17.1.8 Number Sense And Operations

3029 Wikipedia entry.

3030 http://en.wikipedia.org/wiki/Number_sense

3031 Wikipedia entry.

3032 [http://en.wikipedia.org/wiki/Operation_\(mathematics\)](http://en.wikipedia.org/wiki/Operation_(mathematics))

3033 (In development...)

3034 17.1.8.1 Express an integer fraction in lowest terms

3035 """

3036 Problem:

3037 Express 90/105 in lowest terms.

3038 Solution:

3039 One way to solve this problem is to factor both the numerator and the
3040 denominator into prime factors, find the common factors, and then divide both
3041 the numerator and denominator by these factors.

3042 """

3043 n = 90

3044 d = 105

3045 print n,n.factor()

3046 print d,d.factor()

3047 |

3048 Numerator: 2 * 3² * 5

3049 Denominator: 3 * 5 * 7

3050 """

3051 It can be seen that the factors 3 and 5 each appear once in both the numerator
3052 and denominator, so we divide both the numerator and denominator by 3*5:

3053 """

3054 n2 = n/(3*5)

3055 d2 = d/(3*5)

3056 print "Numerator2:",n2

3057 print "Denominator2:",d2

3058 |

3059 Numerator2: 6

3060 Denominator2: 7

3061 """

3062 Therefore, 6/7 is 90/105 expressed in lowest terms.

3063 This problem could also have been solved more directly by simply entering
3064 $90/105$ into a cell because rational number objects are automatically reduced to
3065 lowest terms:
3066 $90/105$
3067 $|$
3068 $6/7$

3070 **17.1.9 Polynomial Functions**

3071 Wikipedia entry.
3072 http://en.wikipedia.org/wiki/Polynomial_function
3073 (In development...)

3074 **17.2 Algebra**

3075 Wikipedia entry.
3076 http://en.wikipedia.org/wiki/Algebra_1
3077 (In development...)

3078 **17.2.1 Absolute Value Functions**

3079 Wikipedia entry.
3080 http://en.wikipedia.org/wiki/Absolute_value
3081 (In development...)

3082 **17.2.2 Complex Numbers**

3083 Wikipedia entry.
3084 http://en.wikipedia.org/wiki/Complex_numbers
3085 (In development...)

3086 **17.2.3 Composite Functions**

3087 Wikipedia entry.
3088 http://en.wikipedia.org/wiki/Composite_function
3089 (In development...)

3090 **17.2.4 Conics**

3091 Wikipedia entry.
3092 <http://en.wikipedia.org/wiki/Conics>
3093 (In development...)

3094 **17.2.5 Data Analysis**

3095 Wikipedia entry.

3096 http://en.wikipedia.org/wiki/Data_analysis

3097 (In development...)

3098 **17.2.6 Discrete Mathematics**

3099 Wikipedia entry.

3100 http://en.wikipedia.org/wiki/Discrete_mathematics

3101 (In development...)

3102 **17.2.7 Equations**

3103 Wikipedia entry.

3104 <http://en.wikipedia.org/wiki/Equation>

3105 (In development...)

3106 **17.2.7.1 Express a symbolic fraction in lowest terms**

3107 """

3108 Problem:

3109 Express $(6x^2 - b) / (b - 6ab)$ in lowest terms, where a and b represent
3110 positive integers.

3111 Solution:

3112 """

3113 `var('a,b')`3114 `n = 6*a^2 - a`3115 `d = b - 6 * a * b`3116 `print n`3117 `print " -----"`3118 `print d`3119 `|`3120 `2`3121 `6 a - a`3122 `-----`3123 `b - 6 a b`

3124 """

3125 We begin by factoring both the numerator and the denominator and then looking
3126 for common factors:

3127 """

3128 `n2 = n.factor()`3129 `d2 = d.factor()`3130 `print "Factored numerator:",n2.__repr__()`

```
3131 print "Factored denominator:",d2.__repr__()
3132 |
3133 Factored numerator: a*(6*a - 1)
3134 Factored denominator: -(6*a - 1)*b

3135 """
3136 At first, it does not appear that the numerator and denominator contain any
3137 common factors. If the denominator is studied further, however, it can be seen
3138 that if (1 - 6 a) is multiplied by -1,
3139 (6 a - 1) is the result and this factor is also present
3140 in the numerator. Therefore, our next step is to multiply both the numerator and
3141 denominator by -1:
3142 """
3143 n3 = n2 * -1
3144 d3 = d2 * -1
3145 print "Numerator * -1:",n3.__repr__()
3146 print "Denominator * -1:",d3.__repr__()
3147 |
3148 Numerator * -1: -a*(6*a - 1)
3149 Denominator * -1: (6*a - 1)*b

3150 """
3151 Now, both the numerator and denominator can be divided by (6*a - 1) in order to
3152 reduce each to lowest terms:
3153 """
3154 common_factor = 6*a - 1
3155 n4 = n3 / common_factor
3156 d4 = d3 / common_factor
3157 print n4
3158 print "          ---"
3159 print d4
3160 |
3161          - a
3162          ---
3163          b

3164 """
3165 The problem could also have been solved more directly using a
3166 SymbolicArithmetic object:
3167 """
3168 z = n/d
3169 z.simplify_rational()
3170 |
3171 -a/b
```

3172 **17.2.7.2 Determine the product of two symbolic fractions**

3173 Perform the indicated operation:

3174 """

3175 Since symbolic expressions are usually automatically simplified, all that needs to
3176 be done with this problem is to enter the expression and assign it to a variable:

3177 """

3178 var('y')

3179 $a = (x/(2*y))^2 * ((4*y^2)/(3*x))^3$

3180 #Display the expression in text form:

3181 a

3182 |

3183 $16*y^4/(27*x)$

3184 #Display the expression in traditional form:

3185 show(a)

3186 |

3187 **17.2.7.3 Solve a linear equation for x**

3188 Solve

3189 """

3190 Like terms will automatically be combined when this equation is placed into a
3191 SymbolicEquation object:

3192 """

3193 $a = 5*x + 2*x - 8 == 5*x - 3*x + 7$

3194 a

3195 |

3196 $7*x - 8 == 2*x + 7$

3197 """

3198 First, lets move the x terms to the left side of the equation by subtracting 2x
3199 from each side. (Note: remember that the underscore '_' holds the result of the
3200 last cell that was executed:

3201 """

3202 $_ - 2*x$

3203 |

3204 $5*x - 8 == 7$

3205 """

3206 Next, add 8 to both sides:

```
3207 """
3208 _+8
3209 |
3210 5*x == 15
3211 """
3212 Finally, divide both sides by 5 to determine the solution:
3213 """
3214 _/5
3215 |
3216 x == 3
3217 """
3218 This problem could also have been solved automatically using the solve()
3219 function:
3220 """
3221 solve(a,x)
3222 |
3223 [x == 3]
```

3224 **17.2.7.4 Solve a linear equation which has fractions**

3225 Solve

```
3226 """
3227 The first step is to place the equation into a SymbolicEquation object. It is good
3228 idea to then display the equation so that you can verify that it was entered
3229 correctly:
3230 """
3231 a = (16*x - 13)/6 == (3*x + 5)/2 - (4 - x)/3
3232 a
3233 |
3234 (16*x - 13)/6 == (3*x + 5)/2 - (4 - x)/3
```

3235 """

3236 In this case, it is difficult to see if this equation has been entered correctly when
3237 it is displayed in text form so lets also display it in traditional form:

```
3238 """
3239 show(a)
3240 |
```

3241 """

3242 The next step is to determine the least common denominator (LCD) of the
3243 fractions in this equation so the fractions can be removed:

```
3244 """
3245 lcm([6,2,3])
3246 |
```

3247 6

3248 """

3249 The LCD of this equation is 6 so multiplying it by 6 removes the fractions:

3250 """

3251 $b = a*6$

3252 b

3253 $|$

3254 $16*x - 13 == 6*((3*x + 5)/2 - (4 - x)/3)$

3255 """

3256 The right side of this equation is still in factored form so expand it:

3257 """

3258 $c = b.expand()$

3259 c

3260 $|$

3261 $16*x - 13 == 11*x + 7$

3262 """

3263 Transpose the 11x to the left side of the equals sign by subtracting 11x from the

3264 SymbolicEquation:

3265 """

3266 $d = c - 11*x$

3267 d

3268 $|$

3269 $5*x - 13 == 7$

3270 """

3271 Transpose the -13 to the right side of the equals sign by adding 13 to the

3272 SymbolicEquation:

3273 """

3274 $e = d + 13$

3275 e

3276 $|$

3277 $5*x == 20$

3278 """

3279 Finally, dividing the SymbolicEquation by 5 will leave x by itself on the left side
3280 of the equals sign and produce the solution:

3281 """

3282 $f = e / 5$

3283 f

3284 $|$

3285 $x == 4$

3286 """

```
3287 This problem could have also be solved automatically using the solve() function:
3288 ""
3289 solve(a,x)
3290 |
3291 [x == 4]
```

3292 **17.2.8 Exponential Functions**

3293 Wikipedia entry.
3294 http://en.wikipedia.org/wiki/Exponential_function
3295 (In development...)

3296 **17.2.9 Exponents**

3297 Wikipedia entry.
3298 <http://en.wikipedia.org/wiki/Exponent>
3299 (In development...)

3300 **17.2.10 Expressions**

3301 Wikipedia entry.
3302 [http://en.wikipedia.org/wiki/Expression_\(mathematics\)](http://en.wikipedia.org/wiki/Expression_(mathematics))
3303 (In development...)

3304 **17.2.11 Inequalities**

3305 Wikipedia entry.
3306 <http://en.wikipedia.org/wiki/Inequality>
3307 (In development...)

3308 **17.2.12 Inverse Functions**

3309 Wikipedia entry.
3310 http://en.wikipedia.org/wiki/Inverse_function
3311 (In development...)

3312 **17.2.13 Linear Equations And Functions**

3313 Wikipedia entry.
3314 http://en.wikipedia.org/wiki/Linear_functions
3315 (In development...)

3316 **17.2.14 Linear Programming**

3317 Wikipedia entry.
3318 http://en.wikipedia.org/wiki/Linear_programming

3319 (In development...)

3320 **17.2.15 Logarithmic Functions**

3321 Wikipedia entry.

3322 http://en.wikipedia.org/wiki/Logarithmic_function

3323 (In development...)

3324 **17.2.16 Logistic Functions**

3325 Wikipedia entry.

3326 http://en.wikipedia.org/wiki/Logistic_function

3327 (In development...)

3328 **17.2.17 Matrices**

3329 Wikipedia entry.

3330 [http://en.wikipedia.org/wiki/Matrix_\(mathematics\)](http://en.wikipedia.org/wiki/Matrix_(mathematics))

3331 (In development...)

3332 **17.2.18 Parametric Equations**

3333 Wikipedia entry.

3334 http://en.wikipedia.org/wiki/Parametric_equation

3335 (In development...)

3336 **17.2.19 Piecewise Functions**

3337 Wikipedia entry.

3338 http://en.wikipedia.org/wiki/Piecewise_function

3339 (In development...)

3340 **17.2.20 Polynomial Functions**

3341 Wikipedia entry.

3342 http://en.wikipedia.org/wiki/Polynomial_function

3343 (In development...)

3344 **17.2.21 Power Functions**

3345 Wikipedia entry.

3346 http://en.wikipedia.org/wiki/Power_function

3347 (In development...)

3348 **17.2.22 Quadratic Functions**

3349 Wikipedia entry.

3350 http://en.wikipedia.org/wiki/Quadratic_function
3351 (In development...)

3352 **17.2.23 Radical Functions**

3353 Wikipedia entry.
3354 http://en.wikipedia.org/wiki/Nth_root
3355 (In development...)

3356 **17.2.24 Rational Functions**

3357 Wikipedia entry.
3358 http://en.wikipedia.org/wiki/Rational_function
3359 (In development...)

3360 **17.2.25 Sequences**

3361 Wikipedia entry.
3362 <http://en.wikipedia.org/wiki/Sequence>
3363 (In development...)

3364 **17.2.26 Series**

3365 Wikipedia entry.
3366 http://en.wikipedia.org/wiki/Series_mathematics
3367 (In development...)

3368 **17.2.27 Systems of Equations**

3369 Wikipedia entry.
3370 http://en.wikipedia.org/wiki/System_of_equations
3371 (In development...)

3372 **17.2.28 Transformations**

3373 Wikipedia entry.
3374 [http://en.wikipedia.org/wiki/Transformation_\(geometry\)](http://en.wikipedia.org/wiki/Transformation_(geometry))
3375 (In development...)

3376 **17.2.29 Trigonometric Functions**

3377 Wikipedia entry.
3378 http://en.wikipedia.org/wiki/Trigonometric_function
3379 (In development...)

3380 **17.3 Precalculus And Trigonometry**

3381 Wikipedia entry.

3382 <http://en.wikipedia.org/wiki/Precalculus>

3383 <http://en.wikipedia.org/wiki/Trigonometry>

3384 (In development...)

3385 **17.3.1 Binomial Theorem**

3386 Wikipedia entry.

3387 http://en.wikipedia.org/wiki/Binomial_theorem

3388 (In development...)

3389 **17.3.2 Complex Numbers**

3390 Wikipedia entry.

3391 http://en.wikipedia.org/wiki/Complex_numbers

3392 (In development...)

3393 **17.3.3 Composite Functions**

3394 Wikipedia entry.

3395 http://en.wikipedia.org/wiki/Composite_function

3396 (In development...)

3397 **17.3.4 Conics**

3398 Wikipedia entry.

3399 <http://en.wikipedia.org/wiki/Conics>

3400 (In development...)

3401 **17.3.5 Data Analysis**

3402 Wikipedia entry.

3403 http://en.wikipedia.org/wiki/Data_analysis

3404 (In development...)

3405 **17.3.6 Discrete Mathematics**

3406 Wikipedia entry.

3407 http://en.wikipedia.org/wiki/Discrete_mathematics

3408 (In development...)

3409 **17.3.7 Equations**

3410 Wikipedia entry.

3411 <http://en.wikipedia.org/wiki/Equation>

3412 (In development...)

3413 **17.3.8 Exponential Functions**

3414 Wikipedia entry.

3415 <http://en.wikipedia.org/wiki/Equation>

3416 (In development...)

3417 **17.3.9 Inverse Functions**

3418 Wikipedia entry.

3419 http://en.wikipedia.org/wiki/Inverse_function

3420 (In development...)

3421 **17.3.10 Logarithmic Functions**

3422 Wikipedia entry.

3423 http://en.wikipedia.org/wiki/Logarithmic_function

3424 (In development...)

3425 **17.3.11 Logistic Functions**

3426 Wikipedia entry.

3427 http://en.wikipedia.org/wiki/Logistic_function

3428 (In development...)

3429 **17.3.12 Matrices And Matrix Algebra**

3430 Wikipedia entry.

3431 [http://en.wikipedia.org/wiki/Matrix_\(mathematics\)](http://en.wikipedia.org/wiki/Matrix_(mathematics))

3432 (In development...)

3433 **17.3.13 Mathematical Analysis**

3434 Wikipedia entry.

3435 http://en.wikipedia.org/wiki/Mathematical_analysis

3436 (In development...)

3437 **17.3.14 Parametric Equations**

3438 Wikipedia entry.

3439 http://en.wikipedia.org/wiki/Parametric_equation

3440 (In development...)

3441 17.3.15 Piecewise Functions

3442 Wikipedia entry.

3443 http://en.wikipedia.org/wiki/Piecewise_function

3444 (In development...)

3445 17.3.16 Polar Equations

3446 Wikipedia entry.

3447 http://en.wikipedia.org/wiki/Polar_equation

3448 (In development...)

3449 17.3.17 Polynomial Functions

3450 Wikipedia entry.

3451 http://en.wikipedia.org/wiki/Polynomial_function

3452 (In development...)

3453 17.3.18 Power Functions

3454 Wikipedia entry.

3455 http://en.wikipedia.org/wiki/Power_function

3456 (In development...)

3457 17.3.19 Quadratic Functions

3458 Wikipedia entry.

3459 http://en.wikipedia.org/wiki/Quadratic_function

3460 (In development...)

3461 17.3.20 Radical Functions

3462 Wikipedia entry.

3463 http://en.wikipedia.org/wiki/Nth_root

3464 (In development...)

3465 17.3.21 Rational Functions

3466 Wikipedia entry.

3467 http://en.wikipedia.org/wiki/Rational_function

3468 (In development...)

3469 17.3.22 Real Numbers

3470 Wikipedia entry.

3471 http://en.wikipedia.org/wiki/Real_number

3472 (In development...)

3473 **17.3.23 Sequences**

3474 Wikipedia entry.

3475 <http://en.wikipedia.org/wiki/Sequence>

3476 (In development...)

3477 **17.3.24 Series**

3478 Wikipedia entry.

3479 [http://en.wikipedia.org/wiki/Series_\(mathematics\)](http://en.wikipedia.org/wiki/Series_(mathematics))

3480 (In development...)

3481 **17.3.25 Sets**

3482 Wikipedia entry.

3483 <http://en.wikipedia.org/wiki/Set>

3484 (In development...)

3485 **17.3.26 Systems of Equations**

3486 Wikipedia entry.

3487 http://en.wikipedia.org/wiki/System_of_equations

3488 (In development...)

3489 **17.3.27 Transformations**

3490 Wikipedia entry.

3491 [http://en.wikipedia.org/wiki/Transformation_\(geometry\)](http://en.wikipedia.org/wiki/Transformation_(geometry))

3492 (In development...)

3493 **17.3.28 Trigonometric Functions**

3494 Wikipedia entry.

3495 http://en.wikipedia.org/wiki/Trigonometric_function

3496 (In development...)

3497 **17.3.29 Vectors**

3498 Wikipedia entry.

3499 <http://en.wikipedia.org/wiki/Vector>

3500 (In development...)

3501 **17.4 Calculus**

3502 Wikipedia entry.

3503 <http://en.wikipedia.org/wiki/Calculus>

3504 (In development...)

3505 17.4.1 Derivatives

3506 Wikipedia entry.

3507 <http://en.wikipedia.org/wiki/Derivative>

3508 (In development...)

3509 17.4.2 Integrals

3510 Wikipedia entry.

3511 <http://en.wikipedia.org/wiki/Integral>

3512 (In development...)

3513 17.4.3 Limits

3514 Wikipedia entry.

3515 [http://en.wikipedia.org/wiki/Limit_\(mathematics\)](http://en.wikipedia.org/wiki/Limit_(mathematics))

3516 (In development...)

3517 17.4.4 Polynomial Approximations And Series

3518 Wikipedia entry.

3519 http://en.wikipedia.org/wiki/Convergent_series

3520 (In development...)

3521 17.5 Statistics

3522 Wikipedia entry.

3523 <http://en.wikipedia.org/wiki/Statistics>

3524 (In development...)

3525 17.5.1 Data Analysis

3526 Wikipedia entry.

3527 http://en.wikipedia.org/wiki/Data_analysis

3528 (In development...)

3529 17.5.2 Inferential Statistics

3530 Wikipedia entry.

3531 http://en.wikipedia.org/wiki/Inferential_statistics

3532 (In development...)

3533 17.5.3 Normal Distributions

3534 Wikipedia entry.

3535 http://en.wikipedia.org/wiki/Normal_distribution

3536 (In development...)

3537 **17.5.4 One Variable Analysis**

3538 Wikipedia entry.

3539 <http://en.wikipedia.org/wiki/Univariate>

3540 (In development...)

3541 **17.5.5 Probability And Simulation**

3542 Wikipedia entry.

3543 <http://en.wikipedia.org/wiki/Probability>

3544 (In development...)

3545 **17.5.6 Two Variable Analysis**

3546 Wikipedia entry.

3547 <http://en.wikipedia.org/wiki/Multivariate>

3548 (In development...)

3549 **18 High School Science Problems**

3550 (In development...)

3551 **18.1 Physics**

3552 Wikipedia entry.

3553 <http://en.wikipedia.org/wiki/Physics>

3554 (In development...)

3555 **18.1.1 Atomic Physics**

3556 Wikipedia entry.

3557 http://en.wikipedia.org/wiki/Atomic_physics

3558 (In development...)

3559 **18.1.2 Circular Motion**

3560 Wikipedia entry.

3561 http://en.wikipedia.org/wiki/Circular_motion

3562 (In development...)

3563 **18.1.3 Dynamics**

3564 Wikipedia entry.

3565 [http://en.wikipedia.org/wiki/Dynamics_\(physics\)](http://en.wikipedia.org/wiki/Dynamics_(physics))

3566 (In development...)

3567 **18.1.4 Electricity And Magnetism**

3568 Wikipedia entry.

3569 <http://en.wikipedia.org/wiki/Electricity>

3570 <http://en.wikipedia.org/wiki/Magnetism>

3571 (In development...)

3572 **18.1.5 Fluids**

3573 Wikipedia entry.

3574 <http://en.wikipedia.org/wiki/Fluids>

3575 (In development...)

3576 **18.1.6 Kinematics**

3577 Wikipedia entry.

3578 <http://en.wikipedia.org/wiki/Kinematics>

3579 (In development...)

3580 **18.1.7 Light**

3581 Wikipedia entry.

3582 <http://en.wikipedia.org/wiki/Light>

3583 (In development...)

3584 **18.1.8 Optics**

3585 Wikipedia entry.

3586 <http://en.wikipedia.org/wiki/Optics>

3587 (In development...)

3588 **18.1.9 Relativity**

3589 Wikipedia entry.

3590 <http://en.wikipedia.org/wiki/Relativity>

3591 (In development...)

3592 **18.1.10 Rotational Motion**

3593 Wikipedia entry.

3594 http://en.wikipedia.org/wiki/Rotational_motion

3595 (In development...)

3596 **18.1.11 Sound**

3597 Wikipedia entry.

3598 <http://en.wikipedia.org/wiki/Sound>

3599 (In development...)

3600 **18.1.12 Waves**

3601 Wikipedia entry.

3602 <http://en.wikipedia.org/wiki/Waves>

3603 (In development...)

3604 **18.1.13 Thermodynamics**

3605 Wikipedia entry.

3606 <http://en.wikipedia.org/wiki/Thermodynamics>

3607 (In development...)

3608 **18.1.14 Work**

3609 Wikipedia entry.

3610 http://en.wikipedia.org/wiki/Mechanical_work
3611 (In development...)

3612 **18.1.15 Energy**

3613 Wikipedia entry.
3614 <http://en.wikipedia.org/wiki/Energy>
3615 (In development...)

3616 **18.1.16 Momentum**

3617 Wikipedia entry.
3618 <http://en.wikipedia.org/wiki/Momentum>
3619 (In development...)

3620 **18.1.17 Boiling**

3621 Wikipedia entry.
3622 <http://en.wikipedia.org/wiki/Boiling>
3623 (In development...)

3624 **18.1.18 Buoyancy**

3625 Wikipedia entry.
3626 <http://en.wikipedia.org/wiki/Bouyancy>
3627 (In development...)

3628 **18.1.19 Convection**

3629 Wikipedia entry.
3630 <http://en.wikipedia.org/wiki/Convection>
3631 (In development...)

3632 **18.1.20 Density**

3633 Wikipedia entry.
3634 <http://en.wikipedia.org/wiki/Density>
3635 (In development...)

3636 **18.1.21 Diffusion**

3637 Wikipedia entry.
3638 <http://en.wikipedia.org/wiki/Diffusion>
3639 (In development...)

3640 18.1.22 Freezing

3641 Wikipedia entry.

3642 <http://en.wikipedia.org/wiki/Freezing>

3643 (In development...)

3644 18.1.23 Friction

3645 Wikipedia entry.

3646 <http://en.wikipedia.org/wiki/Friction>

3647 (In development...)

3648 18.1.24 Heat Transfer

3649 Wikipedia entry.

3650 http://en.wikipedia.org/wiki/Heat_transfer

3651 (In development...)

3652 18.1.25 Insulation

3653 Wikipedia entry.

3654 <http://en.wikipedia.org/wiki/Insulation>

3655 (In development...)

3656 18.1.26 Newton's Laws

3657 Wikipedia entry.

3658 http://en.wikipedia.org/wiki/Newtons_laws

3659 (In development...)

3660 18.1.27 Pressure

3661 Wikipedia entry.

3662 <http://en.wikipedia.org/wiki/Pressure>

3663 (In development...)

3664 18.1.28 Pulleys

3665 Wikipedia entry.

3666 <http://en.wikipedia.org/wiki/Pulley>

3667 (In development...)

