

# **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.

## 16 2 Introduction

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

### 25 2.1 What Is A Super Scientific Calculator?

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

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

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

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

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

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

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

50 and here is the same formula in text form:

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

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](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

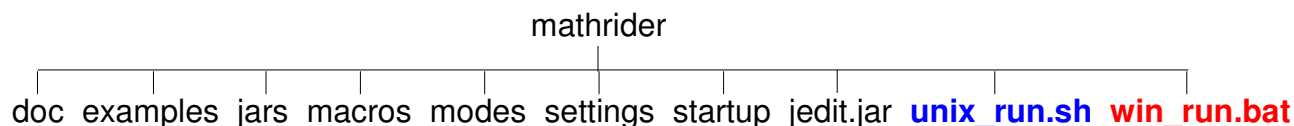
187 One way Unix users can extract the download file is to open a shell, change to  
188 the directory that contains the archive file, and extract it using the following  
189 command:

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 (**expressions** are explained in section [11. MathPiper Programming Fundamentals](#)).  
415 [Fundamentals](#)).

### 416 6.2 MathPiper Program Files

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

### 420 6.3 MathRider Worksheets

421 The most interesting files are MathRider **worksheet** files (which are the ones  
422 that end with a **.mrw** extension). MathRider worksheets consist of **folds** which  
423 contain different types of code that can be executed by pressing  
424 **<shift><enter>** inside of them. Select the **worksheet\_demo\_1.mrw** tab and  
425 follow the instructions which are present within the comments it contains.

### 426 6.4 Plugins

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

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

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

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

440 The Jung plugin was detached from the main window so it can be resized and



441 placed wherever it is needed. Select the inverted black triangle on the floating  
442 windows and try docking the Jung plugin back to the main window again,  
443 perhaps in a different position.

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

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

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

## 451 **7 MathPiper: A Computer Algebra System For Beginners**

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

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

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

### 470 **7.1 Numeric Vs. Symbolic Computations**

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

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

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

482     What makes a symbolic computing system distinct from a non-symbolic (or  
483     numeric) one? We can give one general characterization: the questions one  
484     asks and the resulting answers one expects, are irregular in some way. That  
485     is, their "complexity" may be larger and their sizes may be unpredictable. For  
486     example, if one somehow asks a numeric program to "solve for x in the  
487     equation  $\sin(x) = 0$ " it is plausible that the answer will be some 32-bit  
488     quantity that we could print as 0.0. There is generally no way for such a  
489     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.

527 In addition to addition, MathPiper can also do subtraction, multiplication,  
528 exponents, and division:

529 In> 5-2  
530 Result> 3

531 In> 3\*4  
532 Result> 12

533 In> 2^3  
534 Result> 8

535 In> 12/6  
536 Result> 2

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

541 MathPiper can also work with decimal numbers:

542 In> .5+1.2  
543 Result> 1.7

544 In> 3.7-2.6  
545 Result> 1.1

546 In> 2.2\*3.9  
547 Result> 8.58

548 In> 2.2^3  
549 Result> 10.648

550 In> 9.5/3.2  
551 Result> 9.5/3.2

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

555 In> N(9.5/3.2)  
556 Result> 2.96875

### 557 7.1.1.1 Functions

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

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

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

```
567 In> IsEven(4)
568 Result> True
```

```
569 In> IsEven(5)
570 Result> False
```

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

#### 575 **7.1.1.2 Accessing Previous Input And Results**

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

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

```
582 In> 5*8
583 Result> 40
```

```
584 In> %
585 Result> 40
```

```
586 In> %*2
587 Result> 80
```

#### 588 **7.1.1.3 Syntax Errors**

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

594 In> 12 \ 6

595 Error parsing expression, near token \

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

599 This section provided a short introduction to using MathPiper as a numeric  
600 calculator and the next section contains a short introduction to using MathPiper  
601 as a symbolic calculator.

## 602 7.1.2 Using The MathPiper Console As A Symbolic Calculator

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

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

608 In> 1/2 + 1/3  
609 Result> 5/6

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

614 In> %  
615 Result> 5/6

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

618 In> Numer(%)  
619 Result> 5

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

623 **7.1.2.1 Variables**

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

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

```
634 In> a := 1/2 + 1/3
635 Result> 5/6
```

```
636 In> a
637 Result> 5/6
```

```
638 In> Numer(a)
639 Result> 5
```

```
640 In> Denom(a)
641 Result> 6
```

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

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

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

```
650 In> a := 9
651 Result> 9
```

```
652 In> a
653 Result> 9
```

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

```
656 In> Clear(a)
657 Result> True
```

```
658 In> a
659 Result> a
```

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

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

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

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

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

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

```
676 In> x
677 Result> x
```

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

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

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

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

687 What is more interesting, however, are the results returned by **Factor()** and  
688 **Expand()**. **Factor()** is able to determine when expressions with unbound  
689 variables are sent to it and it uses the rules of algebra to **manipulate** them into  
690 factored form. The **Expand()** function was then able to take the factored  
691 expression  $(x+20)(x+4)$  and manipulate it until it was expanded. One way to



692 remember what the functions **Factor()** and **Expand()** do is to look at the second  
693 letters of their names. The '**a**' in **Factor** can be thought of as **adding**  
694 parentheses to an expression and the '**x**' in **Expand** can be thought of **xing** out  
695 or removing parentheses from an expression.

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

## 700 8 The MathPiper Documentation Plugin

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

### 706 8.1 Function List

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

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

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

### 730 8.2 Mini Web Browser Interface

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

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

## 741 **9 Using MathRider As A Programmer's Text Editor**

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

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

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

### 754 **9.1 Creating, Opening, And Saving Text Files**

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

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

### 767 **9.2 Editing Files**

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

#### 773 **9.2.1 Rectangular Selection Mode**

774 One capability that MathRider has that a word process may not have is the  
775 ability to select rectangular sections of text. To see how this works, do the  
776 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.

## 816 10 MathRider Worksheet Files

817 While MathRider's ability to execute code with consoles and programs provide a  
818 significant amount of power to the user, most of MathRider's power is derived  
819 from **worksheets**. MathRider worksheets are text files which have a **.mrw**  
820 extension and are able to execute multiple types of code in a single text area.  
821 The **worksheet\_demo\_1.mrw** file (which is preloaded in the MathRider  
822 environment when it is first launched) demonstrates how a worksheet is able to  
823 execute multiple types of code in what are called **code folds**.

### 824 10.1 Code Folds

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

```
831 1:%mathpiper  
832 2:  
833 3:"Hello World!";  
834 4:  
835 5:%/mathpiper
```

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

```
840 1:%mathpiper  
841 2:  
842 3:"Hello World!";  
843 4:  
844 5:%/mathpiper  
845 6:  
846 7:    %output,preserve="false"  
847 8:    Result: "Hello World!"  
848 9:    %/output
```

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

854 new version during the next execution of its parent.

## 855 **10.2 Fold Properties**

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

```

860 1:%mathpiper,output="pretty"
861 2:
862 3:x^2 + x/2 + 3;
863 4:
864 5:%/mathpiper
865 6:
866 7:    %output,preserve="false"
867 8:    Result: True
868 9:
869 10:    Side effects:
870 11:
871 12:      2    x
872 13:     x  + - + 3
873 14:         2
874 15:    %/output
  
```

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

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

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

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

```

887 1:%mathpiper,name="example_1",output="pretty"
888 2:
889 3:x^2 + x/2 + 3;
890 4:
891 5:%/mathpiper
  
```

```

892 6:
893 7:   %output,preserve="false"
894 8:   Result: True
895 9:
896 10:   Side effects:
897 11:
898 12:       2    x
899 13:     x  +  -  + 3
900 14:           2
901 15: %/output

```

### 902 10.3 Currently Implemented Fold Types And Properties

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

#### 905 10.3.1 %geogebra & %geogebra\_xml.

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

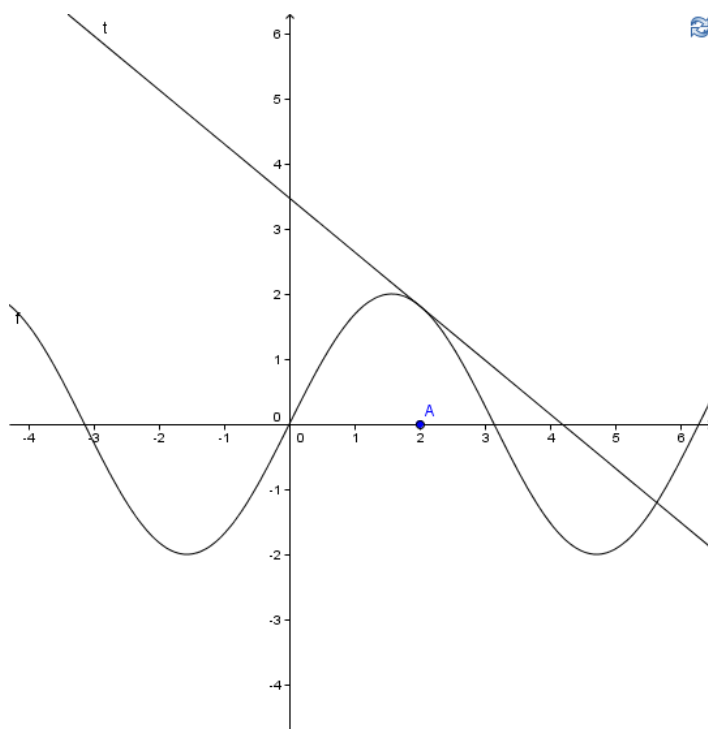
```

912 1: %geogebra,clear="true"
913 2:
914 3: //Plot a function.
915 4: f(x)=2*sin(x)
916 5:
917 6: //Add a tangent line to the function.
918 7: a = 2
919 8: (2,0)
920 9: t = Tangent[a, f]
921 10:
922 11: %/geogebra
923 12:
924 13:   %output,preserve="false"
925 14:   GeoGebra updated.
926 15:   %/output

```

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





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

930 GeoGebra saves information in **.ggb** files and these files are compressed **zip** files  
 931 which have an **XML** file inside of them. The following XML code was obtained by  
 932 adding color information to the previous example, saving it, and unzipping the  
 933 .ggb files that was created. The code was then pasted into a **%geogebra\_xml**  
 934 fold:

```

935 1: %geogebra_xml,description="Obtained from .ggb file"
936 2:
937 3: <?xml version="1.0" encoding="utf-8"?>
938 4: <geogebra format="3.0">
939 5: <gui>
940 6:   <show algebraView="true" auxiliaryObjects="true"
941   algebraInput="true" cmdList="true"/>
942 7:   <splitDivider loc="196" locVertical="400" horizontal="true"/>
943 8:   <font size="12"/>
944 9: </gui>
945 10: <euclidianView>
946 11:   <size width="540" height="553"/>
947 12:   <coordSystem xZero="215.0" yZero="315.0" scale="50.0"
948   yscale="50.0"/>
949 13:   <evSettings axes="true" grid="true" pointCapturing="3"
950   pointStyle="0" rightAngleStyle="1"/>
951 14:   <bgColor r="255" g="255" b="255"/>
952 15:   <axesColor r="0" g="0" b="0"/>

```

```

953 16:    <gridColor r="192" g="192" b="192"/>
954 17:    <lineStyle axes="1" grid="10"/>
955 18:    <axis id="0" show="true" label="" unitLabel="" tickStyle="1"
956    showNumbers="true"/>
957 19:    <axis id="1" show="true" label="" unitLabel="" tickStyle="1"
958    showNumbers="true"/>
959 20:    <grid distX="0.5" distY="0.5"/>
960 21: </euclidianView>
961 22: <kernel>
962 23:    <continuous val="true"/>
963 24:    <decimals val="2"/>
964 25:    <angleUnit val="degree"/>
965 26:    <coordStyle val="0"/>
966 27: </kernel>
967 28: <construction title="" author="" date="">
968 29: <expression label="f" exp="f(x) = 2 sin(x)"/>
969 30: <element type="function" label="f">
970 31:    <show object="true" label="true"/>
971 32:    <objColor r="0" g="0" b="255" alpha="0.0"/>
972 33:    <labelMode val="0"/>
973 34:    <animation step="0.1"/>
974 35:    <fixed val="false"/>
975 36:    <breakpoint val="false"/>
976 37:    <lineStyle thickness="2" type="0"/>
977 38: </element>
978 39: <element type="numeric" label="a">
979 40:    <value val="2.0"/>
980 41:    <show object="false" label="true"/>
981 42:    <objColor r="0" g="0" b="0" alpha="0.1"/>
982 43:    <labelMode val="1"/>
983 44:    <animation step="0.1"/>
984 45:    <fixed val="false"/>
985 46:    <breakpoint val="false"/>
986 47: </element>
987 48: <element type="point" label="A">
988 49:    <show object="true" label="true"/>
989 50:    <objColor r="0" g="0" b="255" alpha="0.0"/>
990 51:    <labelMode val="0"/>
991 52:    <animation step="0.1"/>
992 53:    <fixed val="false"/>
993 54:    <breakpoint val="false"/>
994 55:    <coords x="2.0" y="0.0" z="1.0"/>
995 56:    <coordStyle style="cartesian"/>
996 57:    <pointSize val="3"/>
997 58: </element>
998 59: <command name="Tangent">
999 60:    <input a0="a" a1="f"/>
1000 61:    <output a0="t"/>
1001 62: </command>
1002 63: <element type="line" label="t">

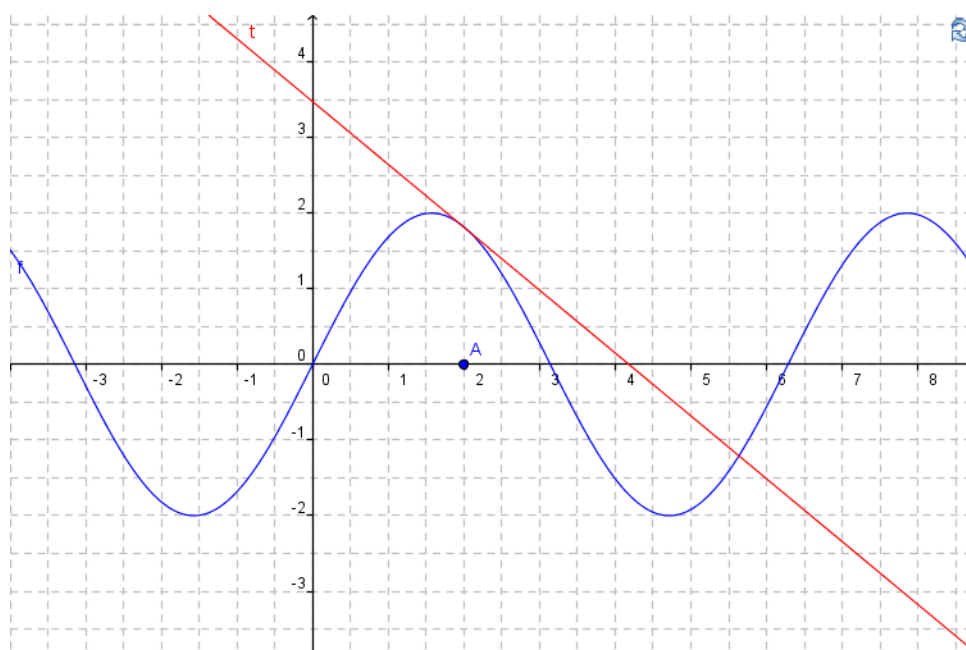
```

```

1003 64: <show object="true" label="true"/>
1004 65: <objColor r="255" g="0" b="0" alpha="0.0"/>
1005 66: <labelMode val="0"/>
1006 67: <breakpoint val="false"/>
1007 68: <coords x="0.8322936730942848" y="1.0" z="-3.4831821998399333"/>
1008 69: <lineStyle thickness="2" type="0"/>
1009 70: <eqnStyle style="explicit"/>
1010 71: </element>
1011 72: </construction>
1012 73: </geogebra>
1013 74:
1014 75: %/geogebra_xml
1015 76:
1016 77: %output,preserve="false"
1017 78:   GeoGebra updated.
1018 79: %/output

```

1019 Illustration 3 shows the result of sending this XML code to GeoGebra:



*Illustration 3: Generated from %geogebra\_xml fold.*

1020 **%geogebra\_xml** folds are not as easy to work with as plain **%geogebra** folds,  
 1021 but they have the advantage of giving the user full control over the GeoGebra  
 1022 environment. Both types of folds can be used together while working with  
 1023 GeoGebra and this means that the user can send code to the GeoGebra plugin  
 1024 from multiple folds during a work session.

### 1025 10.3.2 %hoteqn

1026 Before understanding what the HotEqn (<http://www.atp.ruhr-uni->

1027 [bochum.de/VCLab/software/HotEqn/HotEqn.html](http://bochum.de/VCLab/software/HotEqn/HotEqn.html)) plugin does, one must first  
 1028 know a little bit about LaTeX. LaTeX is a **markup language** which allows  
 1029 formatting information (such as font size, color, and italics) to be added to plain  
 1030 text. LaTeX was designed for creating technical documents and therefore it is  
 1031 capable of marking up mathematics-related text. The hoteqn plugin accepts  
 1032 input marked up with LaTeX's mathematics-oriented commands and displays it in  
 1033 **traditional mathematics** form. For example, to have HotEqn show  $2^3$ , send it  
 1034  $2^{\{3\}}$ :

```
1035 1:%hoteqn
1036 2:
1037 3:2^{\{3\}}
1038 4:
1039 5:%/hoteqn
1040 6:
1041 7:    %output,preserve="false"
1042 8:    HotEqn updated.
1043 9:    %/output
```

1044 and it will display:

$$2^3$$

1045 To have HotEqn show  $2x^3 + 14x^2 + \frac{24x}{7}$ , send it the following code:

```
1046 1:%hoteqn
1047 2:
1048 3:2 x ^{\{3\}} + 14 x ^{\{2\}} + \frac{24 x}{7}
1049 4:
1050 5:%/hoteqn
1051 6:
1052 7:    %output,preserve="false"
1053 8:    HotEqn updated.
1054 9:    %/output
```

1055 and it will display:

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

1056 %hoteqn folds are handy for displaying typed-in LaTeX text in traditional form,  
 1057 but their main use is to allow other folds to display mathematical objects in  
 1058 traditional form. The next section discusses this second use further.

### 1059 10.3.3 %mathpiper

1060 %mathpiper folds were introduced in a previous section and later sections  
1061 discuss how to start programming in MathPiper. This section shows how  
1062 properties can be used to tell %mathpiper folds to generate output that can be  
1063 sent to plugins.

#### 1064 10.3.3.1 Plotting MathPiper Functions With GeoGebra

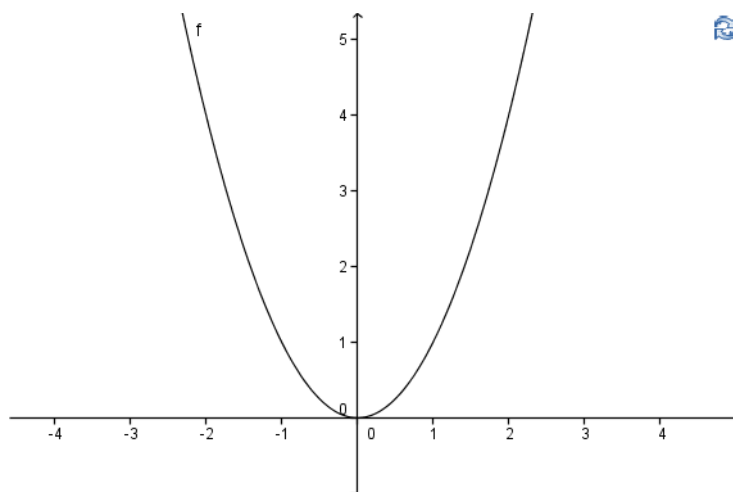
1065 When working with a computer algebra system, a user often needs to plot a  
1066 function in order to understand it better. GeoGebra can plot functions and a  
1067 %mathpiper fold can be configured to generate an executable %geogebra fold by  
1068 setting its **output** property to **geogebra**:

```
1069 1:%mathpiper,output="geogebra"  
1070 2:  
1071 3:x^2;  
1072 4:  
1073 5:%/mathpiper
```

1074 Executing this fold will produce the following output:

```
1075 1:%mathpiper,output="geogebra"  
1076 2:  
1077 3:x^2;  
1078 4:  
1079 5:%/mathpiper  
1080 6:  
1081 7:    %geogebra  
1082 8:    Result: x^2  
1083 9:    %/geogebra
```

1084 Executing the generated %**geogebra** fold will produce an %output fold which  
1085 tells the user that GeoGebra was updated and it will also send the function to the  
1086 GeoGebra plugin for plotting. Illustration 4 shows the plot that was displayed:



*Illustration 4: MathMathPiper Function  
Plotted With GeoGebra*

### 1087 **10.3.3.2 Displaying MathPiper Expressions In Traditional Form With HotEqn**

1088 Reading mathematical expressions in text form is often difficult. Being able to  
 1089 view these expressions in traditional form when needed is helpful and a  
 1090 %mathpiper fold can be configured to do this by setting its output property to  
 1091 **latex**. When the fold is executed, it will generate an executable %hoteqn  
 1092 fold that contains a MathPiper expression which has been converted into a LaTeX  
 1093 expression. The %hoteqn fold can then be executed to view the expression in  
 1094 traditional form:

```

1095 1:%mathpiper,output="latex"
1096 2:
1097 3:((2*x)*(x+3)*(x+4))/9;
1098 5:
1099 6:%/mathpiper
1100 7:
1101 8:    %hoteqn
1102 9:    Result: \frac{2 x \left( x + 3\right) \left( x + 4\right) }{9}
1103 1:    %/hoteqn
1104 2:
1105 3:    %output,preserve="false"
1106 4:    HotEqn updated.
1107 5:    %/output
  
```

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

### 1108 10.3.4 %output

1109 %output folds simply displays text output that has been generated by a parent  
1110 fold. It is not executable and therefore it is not highlighted in light blue like  
1111 executable folds are.

### 1112 10.3.5 %error

1113 %error folds display error messages that have been sent by the software that  
1114 was executing the code in a fold.

### 1115 10.3.6 %html

1116 %html folds display HTML code in a floating window as shown in the following  
1117 example:

```
1118 1: %html,x_size="700",y_size="440"
1119 2:
1120 3: <html>
1121 4:   <h1 align="center">HTML Color Values</h1>
1122 5:   <table border="0" cellpadding="10" cellspacing="1" width="600">
1123 6:     <tr>
1124 7:       <th bgcolor="white" colspan="2"></th>
1125 8:       <th colspan="6">where blue=cc</th>
1126 9:     </tr>
1127 10:    <tr>
1128 11:      <th rowspan="6">where&nbsp;red=</th>
1129 12:      <th>ff</th>
1130 13:      <th bgcolor="#ff00cc">ff00cc</th>
1131 14:      <th bgcolor="#ff33cc">ff33cc</th>
1132 15:      <th bgcolor="#ff66cc">ff66cc</th>
1133 16:      <th bgcolor="#ff99cc">ff99cc</th>
1134 17:      <th bgcolor="#ffcccc">ffcccc</th>
1135 18:      <th bgcolor="#ffffff">ffffcc</th>
1136 19:    </tr>
1137 20:    <tr>
1138 21:      <th>cc</th>
1139 22:      <th bgcolor="#cc00cc">cc00cc</th>
1140 23:      <th bgcolor="#cc33cc">cc33cc</th>
1141 24:      <th bgcolor="#cc66cc">cc66cc</th>
1142 25:      <th bgcolor="#cc99cc">cc99cc</th>
1143 26:      <th bgcolor="#cccccc">cccccc</th>
1144 27:      <th bgcolor="#ccffcc">ccffcc</th>
1145 28:    </tr>
1146 29:    <tr>
1147 30:      <th>99</th>
1148 31:      <th bgcolor="#9900cc">
1149 32:        <font color="#ffffff">9900cc</font>
```

```

1150 33:         </th>
1151 34:         <th bgcolor="#9933cc">9933cc</th>
1152 35:         <th bgcolor="#9966cc">9966cc</th>
1153 36:         <th bgcolor="#9999cc">9999cc</th>
1154 37:         <th bgcolor="#99cccc">99cccc</th>
1155 38:         <th bgcolor="#99ffcc">99ffcc</th>
1156 39:     </tr>
1157 40:     <tr>
1158 41:         <th>66</th>
1159 42:         <th bgcolor="#6600cc">
1160 43:             <font color="#ffffff">6600cc</font>
1161 44:         </th>
1162 45:         <th bgcolor="#6633cc">
1163 46:             <font color="#FFFFFF">6633cc</font>
1164 47:         </th>
1165 48:         <th bgcolor="#6666cc">6666cc</th>
1166 49:         <th bgcolor="#6699cc">6699cc</th>
1167 50:         <th bgcolor="#66cccc">66cccc</th>
1168 51:         <th bgcolor="#66ffcc">66ffcc</th>
1169 52:     </tr>
1170 53:     <tr>
1171 54:         <th colspan="1"></th>
1172 55:         <th>00</th>
1173 56:         <th>33</th>
1174 57:         <th>66</th>
1175 58:         <th>99</th>
1176 59:         <th>cc</th>
1177 60:         <th>ff</th>
1178 61:     </tr>
1179 62:     <tr>
1180 63:         <th colspan="2"></th>
1181 64:         <th colspan="4">where green=</th>
1182 65:     </tr>
1183 66: </table>
1184 67: </html>
1185 68:
1186 69: %/html
1187 70:
1188 71: %output,preserve="false"
1189 72:
1190 73: %/output
1191 74:

```

1192 This code produces the following output:



## HTML Color Values

where blue=cc

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

where red=

where green=

1193 The %html fold's **width** and **height** properties determine the size of the display  
 1194 window.

### 1195 10.3.7 %beanshell

1196 BeanShell (<http://beanshell.org>) is a scripting language that uses Java syntax.  
 1197 MathRider uses BeanShell as its primary customization language and %beanshell  
 1198 folds give MathRider worksheets full access to the internals of MathRider along  
 1199 with the functionality provided by plugins. %beanshell folds are an advanced  
 1200 topic that will be covered in later books.

## 1201 10.4 Automatically Inserting Folds & Removing Unpreserved Folds

1202 Typing the the top and bottom fold lines (for example: %mathpiper ...  
 1203 %/mathpiper) can be tedious and MathRider has a way to automatically insert  
 1204 them. Place the cursor on a line in a .mrw worksheet file where you would like a  
 1205 fold inserted and then **press the right mouse button**. A popup menu will be  
 1206 displayed which will allow you to have a fold automatically inserted into the  
 1207 worksheet at position of the cursor.

1208 This popup menu also has a menu item called "**Remove Unpreserved Folds**". If  
 1209 this menu item is selected, all folds which have a "**preserve="false"**" property  
 1210 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

1244 **division** operator, % is the **remainder** operator, and ^ is the **exponent**  
1245 operator. MathPiper has more operators in addition to these and some of them  
1246 will be covered later.

1247 The following examples show the −, \*, /, %, and ^ operators being used:

1248 In> 5 - 2  
1249 Result> 3

1250 In> 3\*4  
1251 Result> 12

1252 In> 30/3  
1253 Result> 10

1254 In> 8%5  
1255 Result> 3

1256 In> 2^3  
1257 Result> 8

1258 The − character can also be used to indicate a negative number:

1259 In> -3  
1260 Result> -3

1261 Subtracting a negative number results in a positive number:

1262 In> - -3  
1263 Result> 3

1264 In MathPiper, **operators** are symbols (or groups of symbols) which are  
1265 implemented with **functions**. One can either call the function an operator  
1266 represents directly or use the operator to call the function indirectly. However,  
1267 using operators requires less typing and they often make a program easier to  
1268 read.

### 1269 **11.3 Operator Precedence**

1270 When expressions contain more than 1 operator, MathPiper uses a set of rules  
1271 called **operator precedence** to determine the order in which the operators are  
1272 applied to the values in the expression. Operator precedence is also referred to  
1273 as the **order of operations**. Operators with higher precedence are evaluated  
1274 before operators with lower precedence. The following table shows a subset of  
1275 MathPiper's operator precedence rules with higher precedence operators being  
1276 placed higher in the table:

1277      ^      Exponents are evaluated right to left.

```

1278    *,%,/ Then multiplication, remainder, and division operations are evaluated
1279    left to right.

```

1280      +, - Finally, addition and subtraction are evaluated left to right.

1281 Lets manually apply these precedence rules to the multi-operator expression we  
1282 used earlier. Here is the expression in source code form:

1283	$5 + 6*21/18 - 2^3$
------	---------------------

1284 And here it is in traditional form:

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

1285 According to the precedence rules, this is the order in which MathPiper  
1286 evaluates the operations in this expression:

1287     $5 + 6 * 21 / 18 - 2^3$

$$1288 \quad 5 + 6 * 21 / 18 - 8$$

1289    5 + **126**/18 - 8

1290    5 + 7 - 8

1291    **12** - 8

1292 **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**.

1297 **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:

1303 In> 2 + 4\*5

```
1304 Result> 22
```

1305 If parentheses are placed around  $4 + 5$ , however, the addition operator is forced  
1306 to be evaluated before the multiplication operator and the result is 30:

1307 In> (2 + 4)\*5  
1308 Result> 30

1309 Parentheses can also be nested and nested parentheses are evaluated from the  
1310 most deeply nested parentheses outward:

1311 In> ((2 + 4)\*3)\*5  
1312 Result> 90

1313 Since parentheses are evaluated before any other operators, they are placed at  
1314 the top of the precedence table:

- 1315     ()     Parentheses are evaluated from the inside out.
- 1316     ^     Then exponents are evaluated right to left.
- 1317     \*,%,/ Then multiplication, remainder, and division operations are evaluated  
1318           left to right.
- 1319     +, − Finally, addition and subtraction are evaluated left to right.

## 1320 **11.5 Variables**

1321 As discussed in section [7.1.2.1](#), variables are symbols that can be associated with  
1322 values. One way to create variables in MathPiper is through **assignment** and  
1323 this consists of placing the name of a variable you would like to create on the left  
1324 side of an assignment operator **:=** and an expression on the right side of this  
1325 operator. When the expression returns a value, the value is assigned (or **bound**  
1326 to) to the variable.

1327 In the following example, a variable called **box** is created and the number **7** is  
1328 assigned to it:

1329 In> box := 7  
1330 Result> 7

1331 Notice that the assignment operator returns the value that was bound to the  
1332 variable as its result. If you want to see the value that the variable box (or any  
1333 variable) has been bound to, simply evaluate it:

1334 In> box  
1335 Result> 7

1336 If a variable has not been bound to a value yet, it will return itself as the result  
1337 when it is evaluated:

```
1338 In> box2
1339 Result> box2
```

1340 MathPiper variables are **case sensitive**. This means that MathPiper takes into  
1341 account the **case** of each letter in a variable name when it is deciding if two or  
1342 more variable names are the same variable or not. For example, the variable  
1343 name **Box** and the variable name **box** are not the same variable because the first  
1344 variable name starts with an upper case 'B' and the second variable name starts  
1345 with a lower case 'b'.

1346 Programs are able to have more than 1 variable and here is a more sophisticated  
1347 example which uses 3 variables:

```
1348 a := 2
1349 Result> 2
```

```
1350 b := 3
1351 Result> 3
```

```
1352 a + b
1353 Result> 5
```

```
1354 answer := a + b
1355 Result> 5
```

```
1356 answer
1357 Result> 5
```

1358 The part of an expression that is on the right side of an assignment operator is  
1359 always evaluated first and the result is then assigned to the variable that is on  
1360 the left side of the operator.

## 1361 **11.6 Functions & Function Names**

1362 In programming, **functions** are named blocks of code that can be executed one  
1363 or more times by being **called** from other parts of the same program or called  
1364 from other programs. Functions can have values passed to them from the calling  
1365 code and they always return a value back to the calling code when they are  
1366 finished executing. An example of a function is the **IsEven()** function which was  
1367 discussed in an previous section.

1368 Functions are one way that MathPiper enables code to be reused. Most  
1369 programming languages allow code to be reused in this way, although in other  
1370 languages these named blocks of code are sometimes called **subroutines**,  
1371 **procedures**, **methods**, etc.

1372 The functions that come with MathPiper have names which consist of either a  
1373 single word (such as **IsEven()**) or multiple words that have been put together to

1374 form a compound word (such as **IsBound()**). All letters in the names of  
1375 functions which come with MathPiper are lower case except the beginning letter  
1376 in each word, which are upper case.

## 1377 **11.7 Functions That Produce Side Effects**

1378 Most functions are executed to obtain the results they produce but some  
1379 functions are executed in order have them perform work that is not in the form  
1380 of a result. Functions that perform work that is not in the form of a result are  
1381 said to produce **side effects**. Side effects include many forms of work such as  
1382 sending information to the user, opening files, and changing values in memory.

1383 When a function produces a side effect which sends information to the user, this  
1384 information has the words **Side effects:** placed before it instead of the word  
1385 **Result:**. The **Echo()** function is an example of a function that produces a side  
1386 effect and it is covered in the following section.

### 1387 **11.7.1 The Echo() and Write() Functions**

1388 The Echo() and Write() functions both send information to the user and this is  
1389 often referred to as "printing" in this document. It may also be called "echoing"  
1390 and "writing".

#### 1391 **11.7.1.1 Echo()**

1392 The **Echo()** function takes one expression (or multiple expressions separated by  
1393 commas) evaluates each expression, and then prints the results as side effect  
1394 output. The following examples illustrate this:

```
1395 In> Echo(1)
1396 Result> True
1397 Side Effects>
1398 1
```

1399 In this example, the number 1 was passed to the Echo() function, the number  
1400 was evaluated (all numbers evaluate to themselves), and the result of the  
1401 evaluation was then printed as a side effect. Notice that Echo() **also returned a**  
1402 **result**. In MathPiper, all functions return a result but functions whose main  
1403 purpose is to produce a side effect usually just return a result of **True** if the side  
1404 effect succeeded or **False** if it failed. In this case, Echo() returned a result of  
1405 **True** because it was able to successfully print a 1 as its side effect.

1406 The next example shows multiple expressions being sent to Echo() (notice that  
1407 the expressions are separated by commas):

```
1408 In> Echo(1,1+2,2*3)
1409 Result> True
```

```
1410 Side Effects>
1411 1 3 6
```

1412 The expressions were each evaluated and their results were returned as side  
1413 effect output.

1414 Each time an Echo() function is executed, it always forces the display to drop  
1415 down to the next line after it is finished. This can be seen in the following  
1416 program which is similar to the previous one except it uses a separate Echo()  
1417 function to display each expression:

```
1418 1:%mathpiper
1419 2:
1420 3:Echo(1);
1421 4:
1422 5:Echo(1+2);
1423 6:
1424 7:Echo(2*3);
1425 8:
1426 9:%/mathpiper
1427 10:
1428 11:    %output,preserve="false"
1429 12:    Result: True
1430 13:
1431 14:    Side effects:
1432 15:    1
1433 16:    3
1434 17:    6
1435 18:    %/output
```

1436 Notice how the 1, the 3, and the 6 are each on their own line.

1437 Now that we have seen how Echo() works, lets use it to do something useful. If  
1438 more than one expression is evaluated in a %mathpiper fold, only the result from  
1439 the bottommost expression is displayed:

```
1440 1:%mathpiper
1441 2:
1442 3:a := 1;
1443 4:b := 2;
1444 5:c := 3;
1445 6:
1446 7:%/mathpiper
1447 8:
1448 9:    %output,preserve="false"
1449 10:    Result: 3
1450 11:    %/output
```

1451 In MathPiper, programs are executed one line at a time, starting at the topmost



1452 line of code and working downwards from there. In this example, the line `a := 1;`  
1453 is executed first, then the line `b := 2;` is executed, and so on. Notice, however,  
1454 that even though we wanted to see what was in all three variables, only the  
1455 content of the last variable was displayed.

1456 The following example shows how `Echo()` can be used display the contents of all  
1457 three variables:

```
1458 1:%mathpiper
1459 2:
1460 3:a := 1;
1461 4:Echo(a);
1462 5:
1463 6:b := 2;
1464 7:Echo(b);
1465 8:
1466 9:c := 3;
1467 10:Echo(c);
1468 11:
1469 12:%/mathpiper
1470 13:
1471 14:    %output,preserve="false"
1472 15:    Result: True
1473 16:
1474 17:    Side effects:
1475 18:    1
1476 19:    2
1477 20:    3
1478 21:    %/output
```

### 1479 11.7.1.2 Write()

1480 The **Write()** function is similar to the `Echo()` function except it does not  
1481 automatically drop the display down to the next line after it finishes executing:

```
1482 1:%mathpiper
1483 2:
1484 3:Write(1);
1485 4:
1486 5:Write(1+2);
1487 6:
1488 7:Echo(2*3);
1489 8:
1490 9:%/mathpiper
1491 10:
1492 11:    %output,preserve="false"
1493 12:    Result: True
1494 13:
```

```
1495 14:      Side effects:
1496 15:      1 3 6
1497 16:      %/output
```

1498 Write() and Echo() have other differences than the one discussed here and more  
1499 information about them can be found in the documentation for these functions.

## 1500 **11.8 Expressions Are Separated By Semicolons**

1501 In the previous sections, you may have noticed that all of the expressions that  
1502 were executed inside of a **%mathpiper** fold had a semicolon (;) after them but  
1503 the expressions executed in the **MathPiper console** did not have a semicolon  
1504 after them. MathPiper actually requires that all expressions end with a  
1505 semicolon, but one does not need to add a semicolon to an expression which is  
1506 typed into the MathPiper console because the console adds it automatically when  
1507 the expression is executed.

1508 All the previous code examples have had each of their expressions on a separate  
1509 line, but multiple expressions can also be placed on a single line because the  
1510 semicolons tell MathPiper where one expression ends and the next one begins:

```
1511 1:%mathpiper
1512 2:
1513 3:a := 1; Echo(a); b := 2; Echo(b); c := 3; Echo(c);
1514 4:
1515 5:%/mathpiper
1516 6:
1517 7:      %output,preserve="false"
1518 8:      Result: True
1519 9:
1520 10:     Side effects:
1521 11:     1
1522 12:     2
1523 13:     3
1524 14:     %/output
```

1525 The spaces that are in the code on line 2 of this example are used to make the  
1526 code more readable. Any spaces that are present within any expressions or  
1527 between them are ignored by MathPiper and if we removed the spaces from the  
1528 previous code, the output remains the same:

```
1529 1:%mathpiper
1530 2:
1531 3:a:=1;Echo(a);b:=2;Echo(b);c:= 3;Echo(c);
1532 4:
1533 5:%/mathpiper
1534 6:
1535 7:      %output,preserve="false"
```

```
1536 8:      Result: True
1537 9:
1538 10:     Side effects:
1539 11:      1
1540 12:      2
1541 13:      3
1542 14:     %/output
```

## 1543 11.9 Strings

1544 A **string** is a **value** that is used to hold text-based information. The typical  
1545 expression that is used to create a string consists of **text which is enclosed**  
1546 **within double quotes**. Strings can be assigned to variables just like numbers  
1547 can and strings can also be displayed using the Echo() function. The following  
1548 program assigns a string value to the variable 'a' and then echos it to the user:

```
1549 1:%mathpiper
1550 2:
1551 3:a := "Hello, I am a string.";
1552 4:Echo(a);
1553 5:
1554 6:%/mathpiper
1555 7:
1556 8:      %output,preserve="false"
1557 9:      Result: True
1558 10:
1559 11:     Side effects:
1560 12:      Hello, I am a string.
1561 13:     %/output
```

1562 A useful aspect of using MathPiper inside of MathRider is that variables that are  
1563 assigned inside of a **%mathpiper fold** are accessible inside of the **MathPiper**  
1564 **console** and variables that are assigned inside of the **MathPiper console** are  
1565 available inside of **%mathpiper folds**. For example, after the above fold is  
1566 executed, the string that has been bound to variable 'a' can be displayed in the  
1567 MathPiper console:

```
1568 In> a
1569 Result> "Hello, I am a string."
```

1570 Individual characters in a string can be accessed by placing the character's  
1571 position inside of brackets [] after the variable it is assigned. A character's  
1572 position is determined by its distance from the left side of the string, starting at  
1573 1. For example, in the above string, 'H' is at position 1, 'e' is at position 2, etc.  
1574 The following code shows individual characters in the above string being  
1575 accessed:

```
1576 In> a[1]
1577 Result> "H"
```

```
1578 In> a[2]
1579 Result> "e"
```

```
1580 In> a[3]
1581 Result> "l"
```

```
1582 In> a[4]
1583 Result> "l"
```

```
1584 In> a[5]
1585 Result> "o"
```

1586 A range of characters in a string can be accessed by using the .. "range"  
1587 operator:

```
1588 In> a[8 .. 11]
1589 Result> "I am"
```

1590 The .. operator is covered in section [11.17.3.1. The .. Range Operator](#).

## 1591 **11.10 Comments**

1592 Source code can often be difficult to understand and therefore all programming  
1593 languages provide the ability for **comments** to be included in the code.

1594 Comments are used to explain what the code near them is doing and they are  
1595 usually meant to be read by humans instead of being processed by a computer.  
1596 Comments are ignored when the program is executed.

1597 There are two ways that MathPiper allows comments to be added to source code.  
1598 The first way is by placing two forward slashes // to the left of any text that is  
1599 meant to serve as a comment. The text from the slashes to the end of the line  
1600 the slashes are on will be treated as a comment. Here is a program that contains  
1601 comments which use slashes:

```
1602 1:%mathpiper
1603 2://This is a comment.
1604 3:
1605 4:x := 2; //Set the variable x equal to 2.
1606 5:
1607 6:
1608 7:%/mathpiper
1609 8:
1610 9:    %output,preserve="false"
1611 10:    Result: 2
```

1612 11: %/output

1613 When this program is executed, any text that starts with slashes is ignored.

1614 The second way to add comments to a MathPiper program is by enclosing the  
1615 comments inside of slash-asterisk/asterisk-slash symbols `/* */`. This option is  
1616 useful when a comment is too large to fit on one line. Any text between these  
1617 symbols is ignored by the computer. This program shows a longer comment  
1618 which has been placed between these symbols:

```
1619 1:%mathpiper
1620 2:
1621 3:/*
1622 4: This is a longer comment and it uses
1623 5: more than one line. The following
1624 6: code assigns the number 3 to variable
1625 7: x and then returns it as a result.
1626 8:*/
1627 9:
1628 10:x := 3;
1629 11:
1630 12:%/mathpiper
1631 13:
1632 14: %output,preserve="false"
1633 15:     Result: 3
1634 16: %/output
```

## 1635 11.11 Conditional Operators

1636 A conditional operator is an operator that is used to compare two values.  
1637 Expressions that contain conditional operators return a **boolean value** and a  
1638 **boolean value** is one that can either be **True** or **False**. Table 2 shows the  
1639 conditional operators that MathPiper uses:

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

Table 2: Conditional Operators

1640 The following examples show each of the conditional operators in Table 2 being  
 1641 used to compare values that have been assigned to variables **x** and **y**:

```

1642 1:%mathpiper
1643 2:
1644 2:// Example 1.
1645 3:x := 2;
1646 4:y := 3;
1647 5:
1648 6:Echo(x, "=", y, ":", x = y);
1649 7:Echo(x, "!= ", y, ":", x != y);
1650 8:Echo(x, "< ", y, ":", x < y);
1651 9:Echo(x, "<= ", y, ":", x <= y);
1652 10:Echo(x, "> ", y, ":", x > y);
1653 11:Echo(x, ">= ", y, ":", x >= y);
1654 12:
1655 13:%/mathpiper
1656 14:
1657 15:    %output,preserve="false"
1658 16:    Result: True
1659 17:
1660 18:    Side effects:
1661 19:    2 = 3 :False
1662 20:    2 != 3 :True
1663 21:    2 < 3 :True
1664 22:    2 <= 3 :True
1665 23:    2 > 3 :False
1666 24:    2 >= 3 :False
1667 25:    %/output

```

```
1668 1: %mathpiper
1669 2:
1670 3: // Example 2.
1671 4: x := 2;
1672 5: y := 2;
1673 6:
1674 7: Echo(x, "=", y, ":", x = y);
1675 8: Echo(x, "!= ", y, ":", x != y);
1676 9: Echo(x, "< ", y, ":", x < y);
1677 10: Echo(x, "<=", y, ":", x <= y);
1678 11: Echo(x, "> ", y, ":", x > y);
1679 12: Echo(x, ">=", y, ":", x >= y);
1680 13:
1681 14: %/mathpiper
1682 15:
1683 16: %output,preserve="false"
1684 17: Result: True
1685 18:
1686 19: Side effects:
1687 20: 2 = 2 :True
1688 21: 2 != 2 :False
1689 22: 2 < 2 :False
1690 23: 2 <= 2 :True
1691 24: 2 > 2 :False
1692 25: 2 >= 2 :True
1693 25: %/output
```

```
1694 1: %mathpiper
1695 2:
1696 3: // Example 3.
1697 4: x := 3;
1698 5: y := 2;
1699 6:
1700 7: Echo(x, "=", y, ":", x = y);
1701 8: Echo(x, "!= ", y, ":", x != y);
1702 9: Echo(x, "< ", y, ":", x < y);
1703 10: Echo(x, "<=", y, ":", x <= y);
1704 11: Echo(x, "> ", y, ":", x > y);
1705 12: Echo(x, ">=", y, ":", x >= y);
1706 13:
1707 14: %/mathpiper
1708 15:
1709 16: %output,preserve="false"
1710 17: Result: True
1711 18:
1712 19: Side effects:
1713 20: 3 = 2 :False
1714 21: 3 != 2 :True
```

```
1715 22:      3 < 2 :False
1716 23:      3 <= 2 :False
1717 24:      3 > 2 :True
1718 25:      3 >= 2 :True
1719 26:      %/output
```

1720 Conditional operators are placed at a lower level of precedence than the other  
1721 operators we have covered to this point:

1722     ()     Parentheses are evaluated from the inside out.  
1723     ^     Then exponents are evaluated right to left.  
1724     \*,%,/ Then multiplication, remainder, and division operations are evaluated  
1725           left to right.  
1726     +, - Then addition and subtraction are evaluated left to right.  
1727     =,!=,<,<=,>,>= Finally, conditional operators are evaluated.

## 1728 **11.12 Making Decisions With The If() Function & Predicate Expressions**

1729 All programming languages provide the ability to make decisions and the most  
1730 commonly used function for making decisions in MathPiper is the **If()** function.  
1731 There are two calling formats for the If() function:

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

1732 A **predicate** is an expression which evaluates to either **True** or **False**. The way  
1733 the first form of the If() function works is that it evaluates the first expression in  
1734 its argument list (which is the "predicate" expression) and then looks at the value  
1735 that is returned. If this value is **True**, the "then" expression that is listed second  
1736 in the argument list is executed. If the predicate expression evaluates to **False**,  
1737 the "then" expression is not executed.

1738 The following program uses an If() function to determine if the number in  
1739 variable x is greater than 5. If x is greater than 5, the program will echo  
1740 "Greater" and then "End of program":

```
1741 1:%mathpiper
1742 2:
1743 3:x := 6;
1744 4:
1745 5:If(x > 5, Echo(x, "is greater than 5.));
1746 6:
1747 7:Echo("End of program.");
```



```
1748 8:
1749 9: %/mathpiper
1750 10:
1751 11: %output,preserve="false"
1752 12: Result: True
1753 13:
1754 14: Side effects:
1755 15: 6 is greater than 5.
1756 16: End of program.
1757 17: %/output
```

1758 In this program,  $x$  has been set to 6 and therefore the expression  $x > 5$  is **True**.  
1759 When the If() functions evaluates the predicate expression and determines it is  
1760 **True**, it then executes the Echo() function. The second Echo() function at the  
1761 bottom of the program prints "End of program" regardless of what the If()  
1762 function does.

1763 Here is the same program except that  $x$  has been set to **4** instead of **6**:

```
1764 1: %mathpiper
1765 2:
1766 3: x := 4;
1767 4:
1768 5: If(x > 5, Echo(x, "is greater than 5.));
1769 6:
1770 7: Echo("End of program.");
1771 8:
1772 9: %/mathpiper
1773 10:
1774 11: %output,preserve="false"
1775 12: Result: True
1776 13:
1777 14: Side effects:
1778 15: End of program.
1779 16: %/output
```

1780 This time the expression  $x > 4$  returns a value of **False** which causes the If()  
1781 function to not execute the "then" expression that was passed to it.

1782 The second form of the If() function takes a third "else" expression which is  
1783 executed only if the predicate expression is **False**. This program is similar to the  
1784 previous one except an "else" expression has been added to it:

```
1785 1: %mathpiper
1786 2:
1787 3: x := 4;
1788 4:
1789 5: If(x > 5, Echo(x, "is greater than 5."), Echo(x, "is NOT greater than 5.));
1790 6:
```

```
1791 7:Echo("End of program.");
1792 8:
1793 9:%/mathpiper
1794 10:
1795 11:    %output,preserve="false"
1796 12:    Result: True
1797 13:
1798 14:    Side effects:
1799 15:    4 is NOT greater than 5.
1800 16:    End of program.
1801 17:    %/output
```

## 1802 **11.13 The And(), Or(), & Not() Boolean Functions & Infix Notation**

### 1803 **11.13.1 And()**

1804 Sometimes one needs to check if two or more expressions are all **True** and one  
1805 way to do this is with the **And()** function. The And() function has two calling  
1806 formats and this is the first one:

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

1807 This calling format is able to accept one or more expressions as input. If all of  
1808 these expressions returns a value of **True**, the And() function will also return a  
1809 **True**. However, if any of the expressions returns a **False**, then the And()  
1810 function will return a **False**. This can be seen in the following examples:

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

1813 In> And(True, False)
1814 Result> False

1815 In> And(False, True)
1816 Result> False

1817 In> And(True, True, True, True)
1818 Result> True

1819 In> And(True, True, False, True)
1820 Result> False
```

1821 The second format (or **notation**) that can be used to call the And() function is  
1822 called **infix** notation:

```
expression1 And expression2
```

1823 With **infix** notation, an expression is placed on both sides of the And() function  
 1824 name instead of being placed inside of parentheses that are next to it:

```
1825 In> True And True
1826 Result> True
```

```
1827 In> True And False
1828 Result> False
```

```
1829 In> False And True
1830 Result> False
```

1831 Infix notation can only accept two expressions at a time, but it is often more  
 1832 convenient to use than function calling notation. The following program  
 1833 demonstrates using the infix version of the And() function:

```
1834 1:%mathpiper
1835 2:
1836 3:a := 7;
1837 4:b := 9;
1838 5:
1839 6:Echo("1: ", a < 5 And b < 10);
1840 7:Echo("2: ", a > 5 And b > 10);
1841 8:Echo("3: ", a < 5 And b > 10);
1842 9:Echo("4: ", a > 5 And b < 10);
1843 10:
1844 11:If(a > 5 And b < 10, Echo("These expressions are both true.));
1845 12:
1846 13:%/mathpiper
1847 14:
1848 15:    %output,preserve="false"
1849 16:    Result: True
1850 17:
1851 18:    Side effects:
1852 19:    1: False
1853 20:    2: False
1854 21:    3: False
1855 22:    4: True
1856 23:    These expressions are both true.
1857 23:    %/output
```

## 1858 11.13.2 Or()

1859 The Or() function is similar to the And() function in that it has both a function

1860 and an infix calling format and it only works with boolean values. However,  
1861 instead of requiring that all expressions be **True** in order to return a **True**, Or()  
1862 will return a **True** if **one or more expressions are True**.

1863 Here is the function calling format for Or():

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

1864 and these examples show Or() being used with this format:

1865 In> Or(True, False)

1866 Result> True

1867 In> Or(False, True)

1868 Result> True

1869 In> Or(False, False)

1870 Result> False

1871 In> Or(False, False, False, False)

1872 Result> False

1873 In> Or(False, True, False, False)

1874 Result> True

1875 The infix notation format for Or() is as follows:

```
expression1 Or expression2
```

1876 and these examples show this notation being used:

1877 In> True Or False

1878 Result> True

1879 In> False Or True

1880 Result> True

1881 In> False Or False

1882 Result> False

1883 The following program also demonstrates using the infix version of the Or()  
1884 function:

1885 1:%mathpiper

1886 2:

1887 3:a := 7;

```

1888 4:b := 9;
1889 5:
1890 6:Echo("1: ", a < 5 Or b < 10);
1891 7:Echo("2: ", a > 5 Or b > 10);
1892 8:Echo("3: ", a > 5 Or b < 10);
1893 9:Echo("4: ", a < 5 Or b > 10);
1894 10:
1895 11:If(a < 5 Or b < 10,Echo("At least one of these expressions is true.));
1896 12:
1897 13:/mathpiper
1898 14:
1899 15:    %output,preserve="false"
1900 16:    Result: True
1901 17:
1902 18:    Side effects:
1903 19:    1: True
1904 20:    2: True
1905 21:    3: True
1906 22:    4: False
1907 23:    At least one of these expressions is true.
1908 24:    %/output

```

### 11.13.3 Not() & Prefix Notation

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

```
Not(expression)
```

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

```

1915 In> Not(True)
1916 Result> False

1917 In> Not(False)
1918 Result> True

```

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

```
Not expression
```

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

```
1922 In> Not True
1923 Result> False
```

```
1924 In> Not False
1925 Result> True
```

1926 Finally, here is a program that uses the prefix version of Not():

```
1927 1:%mathpiper
1928 2:
1929 3:Echo("3 = 3 is ", 3 = 3);
1930 4:
1931 5:Echo("Not 3 = 3 is ", Not 3 = 3);
1932 6:
1933 7:%/mathpiper
1934 8:
1935 9:    %output,preserve="false"
1936 10:    Result: True
1937 11:
1938 12:    Side effects:
1939 13:    3 = 3 is True
1940 14:    Not 3 = 3 is False
1941 15:    %/output
```

#### 1942 **11.14 The While() Looping Function & Bodied Notation**

1943 Many kinds of machines, including computers, derive much of their power from  
1944 the principle of **repeated cycling**. **Repeated cycling** in a program means to  
1945 execute one or more expressions over and over again and this process is called  
1946 "**looping**". MathPiper provides a number of ways to implement loops in a  
1947 program and these ways range from straight-forward to subtle.

1948 We will begin discussing looping in MathPiper by starting with the straight-  
1949 forward **While** function. The calling format for the **While** function is as follows:

```
1950 While(predicate)
1951 [
1952     body_expressions
1953 ];
```

1954 The **While** function is similar to the **If** function except it will repeatedly execute  
1955 the statements it contains as long as its "predicate" expression it **True**. As soon  
1956 as the predicate expression returns a **False**, the While() function skips the  
1957 expressions it contains and execution continues with the expression that  
1958 immediately follows the While() function (if there is one).

1959 The expressions which are contained in a While() function are called its "**body**"

1960 and all functions which have body expressions are called "**bodied**" functions. If  
1961 a body contains more than one expression then these expressions need to be  
1962 placed within **brackets** []. What body expressions are will become clearer after  
1963 looking a some example programs.

1964 The following program uses a While() function to print the integers from 1 to 10:

```
1965 1:%mathpiper
1966 2:
1967 3:// This program prints the integers from 1 to 10.
1968 4:
1969 5:
1970 6:/*
1971 7:    Initialize the variable x to 1
1972 8:    outside of the While "loop".
1973 9:*/
1974 10:x := 1;
1975 11:
1976 12:While(x <= 10)
1977 13:[
1978 14:    Echo(x);
1979 15:
1980 16:    x := x + 1; //Increment x by 1.
1981 17:];
1982 18:
1983 19:%/mathpiper
1984 20:
1985 21:    %output,preserve="false"
1986 22:    Result: True
1987 23:
1988 24:    Side effects:
1989 25:    1
1990 26:    2
1991 27:    3
1992 28:    4
1993 29:    5
1994 30:    6
1995 31:    7
1996 32:    8
1997 33:    9
1998 34:    10
1999 35:    %/output
```

2000 In this program, a single variable called **x** is created. It is used to tell the Echo()  
2001 function which **integer** to print and it is also used in the expression that  
2002 determines if the While() function should continue to "**loop**" or not.

2003 When the program is executed, 1 is placed into **x** and then the While() function is  
2004 called. The predicate expression **x <= 10** becomes **1 <= 10** and, since 1 is less  
2005 than or equal to 10, a value of **True** is returned by the expression.

2006 The While() function sees that the expression returned a **True** and therefore it  
 2007 executes all of the expressions inside of its **body** from top to bottom.

2008 The Echo() function prints the current contents of x (which is 1) and then the  
 2009 expression  $x := x + 1$ ; is executed.

2010 The expression  $x := x + 1$ ; is a standard expression form that is used in many  
 2011 programming languages. Each time an expression in this form is evaluated, it  
 2012 increases the variable it contains by 1. Another way to describe the effect this  
 2013 expression has on  $x$  is to say that it **increments**  $x$  by **1**.

2014 In this case  $x$  contains **1** and, after the expression is evaluated,  $x$  contains **2**.

2015 After the last expression inside of a While() function is executed, the While()  
 2016 function reevaluates its predicate expression to determine whether it should  
 2017 continue looping or not. Since  $x$  is **2** at this point, the predicate expression  
 2018 returns **True** and the code inside the body of the While() function is executed  
 2019 again. This loop will be repeated until  $x$  is incremented to **11** and the predicate  
 2020 expression returns **False**.

2021 The previous program can be adjusted in a number of ways to achieve different  
 2022 results. For example, the following program prints the integers from 1 to 100 by  
 2023 changing the **10** in the predicate expression to **100**. A Write() function is used in  
 2024 this program so that its output is displayed on the same line until it encounters  
 2025 the wrap margin in MathRider (which can be set in Utilities -> Buffer Options...).

```

2026 1:%mathpiper
2027 2:
2028 3:// Print the integers from 1 to 100.
2029 4:
2030 5:x := 1;
2031 6:
2032 7:While(x <= 100)
2033 8:[
2034 9:   Write(x);
2035 10:
2036 11:   x := x + 1; //Increment x by 1.
2037 12:];
2038 13:
2039 14:%/mathpiper
2040 15:
2041 16:   %output,preserve="false"
2042 17:   Result: True
2043 18:
2044 19:   Side effects:
2045 20:   1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
2046      24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
2047      44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
2048      64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83
2049      84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
2050 21:   %/output

```



2051 The following program prints the odd integers from 1 to 99 by changing the  
2052 increment value in the increment expression from **1** to **2**:

```
2053 1:%mathpiper
2054 2:
2055 3://Print the odd integers from 1 to 99.
2056 4:
2057 5:x := 1;
2058 6:
2059 7:While(x <= 100)
2060 8:[
2061 9:    Write(x);
2062 10:    x := x + 2; //Increment x by 2.
2063 11:];
2064 12:
2065 13:%/mathpiper
2066 14:
2067 15:    %output,preserve="false"
2068 16:    Result: True
2069 17:
2070 18:    Side effects:
2071 19:    1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43
2072    45 47 49 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83
2073    85 87 89 91 93 95 97 99
2074 20:    %/output
```

2075 Finally, the following program prints the numbers from 1 to 100 in reverse order:

```
2076 1:%mathpiper
2077 2:
2078 3://Print the integers from 1 to 100 in reverse order.
2079 4:
2080 5:x := 100;
2081 6:
2082 7:While(x >= 1)
2083 8:[
2084 9:    Write(x);
2085 10:    x := x - 1; //Decrement x by 1.
2086 11:];
2087 12:
2088 13:%/mathpiper
2089 14:
2090 15:    %output,preserve="false"
2091 16:    Result: True
2092 17:
2093 18:    Side effects:
2094 19:    100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82
2095    81 80 79 78 77 76 75 74 73 72 71 70 69 68 67 66 65 64 63
2096    62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44
```

```

2097          43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25
2098          24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4
2099          3 2 1
2100 20:      %/output

```

2101 In order to achieve the reverse ordering, this program had to initialize **x** to **100**,  
 2102 check to see if **x** was **greater than or equal to 1** ( $x \geq 1$ ), and **decrement** **x** by  
 2103 **subtracting 1 from it** instead of adding 1 to it.

## 2104 **11.15 Long-Running Loops, Infinite Loops, & Interrupting Execution**

2105 It is easy to create a loop that will execute a **large number of times**, or even **an**  
 2106 **infinite number of times**, either on purpose or by mistake. When you execute  
 2107 a program that contains an **infinite loop**, it will run until you tell MathPiper to  
 2108 **interrupt** its execution. This is done by selecting the **MathPiper Plugin** (which  
 2109 has been placed near the upper left part of the application) and then pressing the  
 2110 **"Stop Current Calculation"** button which it contains. (**Note: currently this**  
 2111 **button only works if MathPiper is executed inside of a %mathpiper fold.**)

2112 Lets experiment with this button by executing a program that contains an infinite  
 2113 loop and then stopping it:

```

2114 1:%mathpiper
2115 2:
2116 3://Infinite loop example program.
2117 4:
2118 5:x := 1;
2119 6:While(x < 10)
2120 7:[
2121 8:    answer := x + 1;
2122 9:];
2123 10:
2124 11:%/mathpiper
2125 12:
2126 13:    %output,preserve="false"
2127 14:    Processing...
2128 15:    %/output

```

2129 Since the contents of **x** is never changed inside the loop, the expression **x < 10**  
 2130 always evaluates to **True** which causes the loop to continue looping. Notice that  
 2131 the %output fold contains the word **"Processing..."** to indicate that the program  
 2132 is executing the code.

2133 Execute this program now and then interrupt it using the **"Stop Current**  
 2134 **Calculation"** button. When the program is interrupted, the %output fold will  
 2135 display the message **"User interrupted calculation"** to indicate that the  
 2136 program was interrupted.

## 2137 **11.16 Predicate Functions**

2138 A predicate function is a function that either returns **True** or **False**. Most  
2139 predicate functions in MathPiper have their names begin with "Is". For example,  
2140 IsEven(), IsOdd(), IsInteger, etc. The following examples show some of the  
2141 predicate functions that are in MathPiper:

2142 In> IsEven(4)

2143 Result> True

2144 In> IsEven(5)

2145 Result> False

2146 In> IsZero(0)

2147 Result> True

2148 In> IsZero(1)

2149 Result> False

2150 In> IsNegativeInteger(-1)

2151 Result> True

2152 In> IsNegativeInteger(1)

2153 Result> False

2154 In> IsPrime(7)

2155 Result> True

2156 In> IsPrime(100)

2157 Result> False

2158 There is also an IsBound() and an IsUnbound() function that can be used to  
2159 determine whether or not a value is bound to a given variable:

2160 In> a

2161 Result> a

2162 In> IsBound(a)

2163 Result> False

2164 In> a := 1

2165 Result> 1

2166 In> IsBound(a)

2167 Result> True

2168 In> Clear(a)

2169 Result> True

```
2170 In> a
2171 Result> a
```

```
2172 In> IsBound(a)
2173 Result> False
```

### 2174 ***11.17 Lists: Values That Hold Sequences Of Expressions***

2175 The **list** value type is designed to hold expressions in an ordered collection or  
2176 sequence. Lists are very flexible and they are one of the most heavily used value  
2177 types in MathPiper. Lists can hold expressions of any type, they can grow and  
2178 shrink as needed, and they can be nested. Expressions in a list can be accessed  
2179 by their position in the list and they can also be replaced by other expressions.

2180 One way to create a list is by placing zero or more objects or expressions inside  
2181 of a pair of **braces {}**. The following program creates a list that contains  
2182 various expressions and assigns it to the variable x:

```
2183 In> x := {7,42,"Hello",1/2,var}
2184 Result> {7,42,"Hello",1/2,var}
```

```
2185 In> x
2186 Result> {7,42,"Hello",1/2,var}
```

2187 The number of expressions in a list can be determined with the **Length()**  
2188 function:

```
2189 In> Length({7,42,"Hello",1/2,var})
2190 Result> 5
```

2191 A single expression in a list can be accessed by placing a set of **brackets []** to  
2192 the right of the variable and then putting the expression's position number inside  
2193 of the brackets (Notice that the first expression in the list is at position 1  
2194 counting from the left side of the list):

```
2195 In> x[1]
2196 Result> 7
```

```
2197 In> x[2]
2198 Result> 42
```

```
2199 In> x[3]
2200 Result> "Hello"
```

```
2201 In> x[4]
2202 Result> 1/2
```

```
2203 In> x[5]
```

2204 `Result> var`

2205 The **1st** and **2nd** expressions in this list are **integers**, the **3rd** expression is a  
2206 **string**, the **4th** expression is a **rational number** and the **5th** expression is a  
2207 **variable**. Lists can also hold other lists as shown in the following example:

2208 `In> x := {20, 30, {31, 32, 33}, 40}`

2209 `Result> {20,30,{31,32,33},40}`

2210 `In> x[1]`

2211 `Result> 20`

2212 `In> x[2]`

2213 `Result> 30`

2214 `In> x[3]`

2215 `Result> {31,32,33}`

2216 `In> x[4]`

2217 `Result> 40`

2218

2219 The expression in the **3rd** position in the list is another **list** which contains the  
2220 expressions **31**, **32**, and **33**. An expression in this second list can be accessed by  
2221 two two sets of brackets:

2222 `In> x[3][2]`

2223 `Result> 32`

2224 The **3** inside of the first set of brackets accesses the **3rd** member of the **first** list  
2225 and the **2** inside of the second set of brackets accesses the **2nd** member of the  
2226 **second** list.

### 2227 11.17.1 Using While() Loops With Lists

2228 Functions that loop can be used to select each expression in a list in turn so that  
2229 an operation can be performed on these expressions. The following program  
2230 uses a While() loop to print each of the expressions in a list:

2231 `1:%mathpiper`

2232 `2:`

2233 `3://Print each in in the list.`

2234 `4:`

2235 `5:x := {55,93,40,21,7,24,15,14,82};`

2236 `6:y := 1;`

2237 `7:`

2238 `8:While(y <= 9)`

2239 `9:[`

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

2260 A **loop** can also be used to search through a list. The following program uses a  
2261 **While()** function and an **If()** function to search through a list to see if it contains  
2262 the number **53**. If 53 is found in the list, a message is printed:

```
2263 1: %mathpiper
2264 2:
2265 3: //Determine if 53 is in the list.
2266 4:
2267 5: testList := {18,26,32,42,53,43,54,6,97,41};
2268 6: index := 1;
2269 7:
2270 8: While(index <= 10)
2271 9: [
2272 10:    If(testList[index] = 53,
2273 11:        Echo("53 was found in the list at position", index));
2274 12:
2275 13:    index := index + 1;
2276 14: ];
2277 15:
2278 16: %/mathpiper
2279 17:
2280 18:    %output,preserve="false"
2281 19:    Result: True
2282 20:
2283 21:    Side effects:
2284 22:    53 was found in the list at position 5
2285 23: %/output
```

2286 When this program was executed, it determined that **53** was present in the list at  
2287 position **5**.

## 2288 11.17.2 The ForEach() Looping Function

2289 The **ForEach()** function uses a **loop** to index through a list like the While()  
2290 function does, but it is more flexible and automatic. ForEach() uses bodied  
2291 notation like the While() function does and here is its calling format:

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

2292 **ForEach()** selects each expression in a list in turn, assigns it to the passed-in  
2293 "variable", and then executes the expressions that are inside of "body".

2294 Therefore, body is executed once for each expression in the list.

2295 This example shows how ForEach() can be used to print all of the items in a list:

```
2296 1:%mathpiper
2297 2:
2298 3://Print all values in a list.
2299 4:
2300 5:ForEach(x, {50,51,52,53,54,55,56,57,58,59})
2301 6:[
2302 7:    Echo(x);
2303 8:];
2304 9:
2305 10:%/mathpiper
2306 11:
2307 12:    %output,preserve="false"
2308 13:    Result: True
2309 14:
2310 15:    Side effects:
2311 16:    50
2312 17:    51
2313 18:    52
2314 19:    53
2315 20:    54
2316 21:    55
2317 22:    56
2318 23:    57
2319 24:    58
2320 25:    59
2321 26:%/output
```

## 2322 **11.18 Functions & Operators Which Loop Internally To Process Lists**

2323 Looping is such a useful capability that MathPiper has many functions which  
2324 loop internally. This section discusses a number of functions that use internal  
2325 loops to process lists.

### 2326 **11.18.1 TableForm()**

```
TableForm(list)
```

2327 The TableForm() function prints the contents of a list in the form of a table. Each  
2328 member in the list is printed on its own line and this makes the contents of the  
2329 list easier to read:

```
2330 In> testList := {2,4,6,8,10,12,14,16,18,20}
```

```
2331 Result> {2,4,6,8,10,12,14,16,18,20}
```

```
2332 In> TableForm(testList)
```

```
2333 Result> True
```

```
2334 Side Effects>
```

```
2335 2
```

```
2336 4
```

```
2337 6
```

```
2338 8
```

```
2339 10
```

```
2340 12
```

```
2341 14
```

```
2342 16
```

```
2343 18
```

```
2344 20
```

### 2345 **11.18.2 The .. Range Operator**

```
first .. last
```

2346 One often needs to create a list of consecutive integers and the .. range operator  
2347 can be used to do this. The first integer in the list is placed before the ..  
2348 operator (with a space in between them) and the last integer in the list is placed  
2349 after the .. operator. Here are some examples:

```
2350 In> 1 .. 10
```

```
2351 Result> {1,2,3,4,5,6,7,8,9,10}
```

```
2352 In> 10 .. 1
```

```
2353 Result> {10,9,8,7,6,5,4,3,2,1}
```



```
2354 In> -10 .. 10
2355 Result> {-10,-9,-8,-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,8,9,10}
```

2356 As the examples show, the `..` operator can generate lists of integers in ascending  
2357 order and descending order. It can also generate lists that contain negative  
2358 integers.

### 2359 11.18.3 Contains()

2360 The **Contains()** function searches a list to determine if it contains a given  
2361 expression. If it finds the expression, it returns **True** and if it doesn't find the  
2362 expression, it returns **False**. Here is the calling format for Contains():

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

2363 The following code shows Contains() being used to locate a number in a list:

```
2364 In> Contains({50,51,52,53,54,55,56,57,58,59}, 53)
2365 Result> True
```

```
2366 In> Contains({50,51,52,53,54,55,56,57,58,59}, 75)
2367 Result> False
```

2368 The **Not()** function can also be used with predicate functions like Contains() to  
2369 change their results:

```
2370 In> Not Contains({50,51,52,53,54,55,56,57,58,59}, 75)
2371 Result> True
```

### 2372 11.18.4 Find()

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

2373 The **Find()** function searches a list for the first occurrence of a given expression.  
2374 If the expression is found, the numerical position of its first occurrence is  
2375 returned and if it is not found, -1 is returned:

```
2376 In> Find({23, 15, 67, 98, 64}, 15)
2377 Result> 2
```

```
2378 In> Find({23, 15, 67, 98, 64}, 8)
2379 Result> -1
```

2380 **11.18.5 Count()**

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

2381 **Count()** determines the number of times a given expression occurs in a list:

2382 In> testList := {a,b,b,c,c,c,d,d,d,d,e,e,e,e,e}

2383 Result> {a,b,b,c,c,c,d,d,d,d,e,e,e,e,e}

2384 In> Count(testList, c)

2385 Result> 3

2386 In> Count(testList, e)

2387 Result> 5

2388 In> Count(testList, z)

2389 Result> 0

2390 **11.18.6 Select()**

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

2391 **Select()** returns a list that contains all the expressions in a list which make a  
2392 given predicate return **True**:

2393 In> Select("IsPositiveInteger",{46,87,59,-27,11,86,-21,-58,-86,-52})

2394 Result> {46,87,59,11,86}

2395 In this example, notice that the **name** of the predicate function is passed to  
2396 Select() in **double quotes**. There are other ways to pass a predicate function to  
2397 Select() but these are covered in a later section.

2398 Here are some further examples which use the Select() function:

2399 In> Select("IsOdd", {16,14,82,92,33,74,99,67,65,52})

2400 Result> {33,99,67,65}

2401 In> Select("IsEven", {16,14,82,92,33,74,99,67,65,52})

2402 Result> {16,14,82,92,74,52}

2403 In> Select("IsPrime", 1 .. 75)

2404 Result> {2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73}

2405 Notice how the third example uses the .. operator to automatically generate a list  
2406 of consecutive integers from 1 to 75 for the Select() function to analyze.

### 2407 11.18.7 The Nth() Function & The [] Operator

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

2408 The **Nth()** function simply returns the expression which is at a given index in a  
2409 list. This example shows the third expression in a list being obtained:

```
2410 In> testList := {a,b,c,d,e,f,g}
```

```
2411 Result> {a,b,c,d,e,f,g}
```

```
2412 In> Nth(testList, 3)
```

```
2413 Result> c
```

2414 As discussed earlier, the **[]** operator can also be used to obtain a single  
2415 expression from a list:

```
2416 In> testList[3]
```

```
2417 Result> c
```

2418 The **[]** operator can even obtain a single expression directly from a list without  
2419 needing to use a variable:

```
2420 In> {a,b,c,d,e,f,g}[3]
```

```
2421 Result> c
```

### 2422 11.18.8 Append() & Nondestructive List Operations

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

2423 The **Append()** function adds an expression to the end of a list:

```
2424 In> testList := {21,22,23}
```

```
2425 Result> {21,22,23}
```

```
2426 In> Append(testList, 24)
```

```
2427 Result> {21,22,23,24}
```

2428 However, instead of changing the **original** list, MathPiper creates a **copy** of the  
2429 **original** list and appends the expression to the **copy**. This can be confirmed by  
2430 evaluating the variable **testList** after the **Append()** function has been called:

```
2431 In> testList
```

```
2432 Result> {21,22,23}
```

2433 Notice that the list that is bound to **testList** was not modified by the Append()  
2434 function. This is called a **nondestructive list operation** and most MathPiper  
2435 functions that manipulate lists do so nondestructively. To have the changed list  
2436 bound to the variable that it being used, the following technique can be  
2437 employed:

```
2438 In> testList := {21,22,23}  
2439 Result> {21,22,23}
```

```
2440 In> testList := Append(testList, 24)  
2441 Result> {21,22,23,24}
```

```
2442 In> testList  
2443 Result> {21,22,23,24}
```

2444 After this code has been executed, the modified list has indeed been bound to  
2445 testList as desired.

2446 There are some functions, such as DestructiveAppend(), which **do** change the  
2447 original list and most of them begin with the word "Destructive". These are  
2448 called "destructive functions" and it is recommended that destructive functions  
2449 should be used with care.

### 2450 11.18.9 The : Prepend Operator

```
expression : list
```

2451 The prepend operator is a colon : and it can be used to add an expression to the  
2452 beginning of a list:

```
2453 In> testList := {b,c,d}  
2454 Result> {b,c,d}
```

```
2455 In> testList := a:testList  
2456 Result> {a,b,c,d}
```

### 2457 11.18.10 Concat()

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

2458 The Concat() function is short for "concatenate" which means to join together  
2459 sequentially. It takes two or more lists and joins them together into a  
2460 single larger list:

```
2461 In> Concat({a,b,c}, {1,2,3}, {x,y,z})
2462 Result> {a,b,c,1,2,3,x,y,z}
```

### 2463 11.18.11 Insert(), Delete(), & Replace()

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

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

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

2464 **Insert()** inserts an expression into a list at a given index, **Delete()** deletes an  
2465 expression from a list at a given index, and **Replace()** replaces an expression in  
2466 a list at a given index with another expression:

```
2467 In> testList := {a,b,c,d,e,f,g}
2468 Result> {a,b,c,d,e,f,g}

2469 In> testList := Insert(testList, 4, 123)
2470 Result> {a,b,c,123,d,e,f,g}

2471 In> testList := Delete(testList, 4)
2472 Result> {a,b,c,d,e,f,g}

2473 In> testList := Replace(testList, 4, xxx)
2474 Result> {a,b,c,xxx,e,f,g}
```

### 2475 11.18.12 Take()

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

2476 **Take()** obtains a sublist from the **beginning** of a list, the **end** of a list, or the  
2477 **middle** of a list. The expressions in the list that are not taken are discarded.  
2478 A **positive** integer passed to Take() indicates how many expressions should be  
2479 taken from the **beginning** of a list:

2480 In> testList := {a,b,c,d,e,f,g}

2481 Result> {a,b,c,d,e,f,g}

2482 In> Take(testList, 3)

2483 Result> {a,b,c}

2484 A **negative** integer passed to Take() indicates how many expressions should be  
2485 taken from the **end** of a list:

2486 In> Take(testList, -3)

2487 Result> {e,f,g}

2488 Finally, if a **two member list** is passed to Take() it indicates the **range** of  
2489 expressions that should be taken from the **middle** of a list. The **first** value in the  
2490 passed-in list specifies the **beginning** index of the range and the **second** value  
2491 specifies its **end**:

2492 In> Take(testList, {3,5})

2493 Result> {c,d,e}

### 2494 11.18.13 Drop()

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

2495 **Drop()** does the opposite of Take() in that it **drops** expressions from the  
2496 **beginning** of a list, the **end** of a list, or the **middle** of a list and **returns a list**  
2497 **which contains the remaining expressions.**

2498 A **positive** integer passed to Drop() indicates how many expressions should be  
2499 dropped from the **beginning** of a list:

2500 In> testList := {a,b,c,d,e,f,g}

2501 Result> {a,b,c,d,e,f,g}

2502 In> Drop(testList, 3)

2503 Result> {d,e,f,g}

2504 A **negative** integer passed to Drop() indicates how many expressions should be  
2505 dropped from the **end** of a list:

2506 In> Drop(testList, -3)

2507 Result> {a,b,c,d}

2508 Finally, if a **two member list** is passed to Drop() it indicates the **range** of  
2509 expressions that should be dropped from the **middle** of a list. The **first** value in  
2510 the passed-in list specifies the **beginning** index of the range and the **second**  
2511 value specifies its **end**:

```
2512 In> Drop(testList, {3,5})  
2513 Result> {a,b,f,g}
```

#### 2514 **11.18.14 FillList()**

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

2515 The FillList() function simply creates a list which is of size "length" and fills it  
2516 with "length" copies of the given expression:

```
2517 In> FillList(a, 5)  
2518 Result> {a,a,a,a,a}  
  
2519 In> FillList(42,8)  
2520 Result> {42,42,42,42,42,42,42,42}
```

#### 2521 **11.18.15 RemoveDuplicates()**

```
RemoveDuplicates(list)
```

2522 **RemoveDuplicates()** removes any duplicate expressions that are contained in  
2523 in a list:

```
2524 In> testList := {a,a,b,c,c,b,b,a,b,c,c}  
2525 Result> {a,a,b,c,c,b,b,a,b,c,c}  
  
2526 In> RemoveDuplicates(testList)  
2527 Result> {a,b,c}
```

#### 2528 **11.18.16 Reverse()**

```
Reverse(list)
```

2529 **Reverse()** reverses the order of the expressions in a list:

2530 In> testList := {a,b,c,d,e,f,g,h}

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

2532 In> Reverse(testList)

2533 Result> {h,g,f,e,d,c,b,a}

### 2534 11.18.17 Partition()

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

2535 The **Partition()** function breaks a list into sublists of size "partition\_size":

2536 In> testList := {a,b,c,d,e,f,g,h}

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

2538 In> Partition(testList, 2)

2539 Result> {{a,b},{c,d},{e,f},{g,h}}

2540 If the partition\_size does not divide the length of the list evenly, the remaining  
2541 elements are discarded:

2542 In> Partition(testList, 3)

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

2544 The number of elements that Partition() will discard can be calculated by  
2545 dividing the length of a list by the partition size and obtaining the remainder:

2546 In> Mod(Length(testList), 3)

2547 Result> 2

2548 The Mod() function, which divides two integers and return their remainder, is  
2549 covered in a later section.

## 2550 11.19 Functions That Work With Integers

2551 This section discusses various functions which work with integers. Some of  
2552 these functions also work with non-integer values and their use with non-  
2553 integers is discussed in other sections.

### 2554 11.19.1 RandomIntegerVector()

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



2555 A vector can be thought of as a list that does not contain other lists.  
2556 **RandomIntegerVector()** creates a list of size "length" that contains random  
2557 integers that are no lower than "lowest\_possible" and no higher than "highest  
2558 possible". The following example creates **10** random integers between **1** and **99**  
2559 inclusive:

```
2560 In> RandomIntegerVector(10, 1, 99)
2561 Result> {73,93,80,37,55,93,40,21,7,24}
```

### 2562 11.19.2 Max() & Min()

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

2563 If two values are passed to Max(), it determines which one is larger:

```
2564 In> Max(10, 20)
2565 Result> 20
```

2566 If a list of values are passed to Max(), it finds the largest value in the list:

```
2567 In> testList := RandomIntegerVector(10, 1, 99)
2568 Result> {73,93,80,37,55,93,40,21,7,24}
```

```
2569 In> Max(testList)
2570 Result> 93
```

2571 The **Min()** function is the opposite of the Max() function.

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

2572 If two values are passed to Min(), it determines which one is smaller:

```
2573 In> Min(10, 20)
2574 Result> 10
```

2575 If a list of values are passed to Min(), it finds the smallest value in the list:

```
2576 In> testList := RandomIntegerVector(10, 1, 99)
2577 Result> {73,93,80,37,55,93,40,21,7,24}
```

```
2578 In> Min(testList)
2579 Result> 7
```

### 2580 11.19.3 Div() & Mod()

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

2581 **Div()** stands for "divide" and determines the whole number of times a divisor  
2582 goes into a dividend:

```
2583 In> Div(7, 3)
2584 Result> 2
```

2585 **Mod()** stands for "modulo" and it determines the remainder that results when a  
2586 dividend is divided by a divisor:

```
2587 In> Mod(7,3)
2588 Result> 1
```

2589 The remainder/modulo operator % can also be used to calculate a remainder:

```
2590 In> 7 % 2
2591 Result> 1
```

### 2592 11.19.4 Gcd()

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

2593 GCD stands for Greatest Common Divisor and the **Gcd()** function determines the  
2594 greatest common divisor of the values that are passed to it.

2595 If two integers are passed to Gcd(), it calculates their greatest common divisor:

```
2596 In> Gcd(21, 56)
2597 Result> 7
```

2598 If a list of integers are passed to Gcd(), it finds the greatest common divisor of all  
2599 the integers in the list:

```
2600 In> Gcd({9, 66, 123})
2601 Result> 3
```

2602 **11.19.5 Lcm()**

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

2603 LCM stands for Least Common Multiple and the **Lcm()** function determines the  
2604 least common multiple of the values that are passed to it.

2605 If two integers are passed to Lcm(), it calculates their least common multiple:

```
2606 In> Lcm(14, 8)
2607 Result> 56
```

2608 If a list of integers are passed to Lcm(), it finds the least common multiple of all  
2609 the integers in the list:

```
2610 In> Lcm({3,7,9,11})
2611 Result> 693
```

2612 **11.19.6 Add()**

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

2613 **Add()** can find the sum of two or values passed to it:

```
2614 In> Add(3,8,20,11)
2615 Result> 42
```

2616 It can also find the sum of a list of values :

```
2617 In> testList := RandomIntegerVector(10,1,99)
2618 Result> {73,93,80,37,55,93,40,21,7,24}
```

```
2619 In> Add(testList)
2620 Result> 523
```

```
2621 In> testList := 1 .. 10
2622 Result> {1,2,3,4,5,6,7,8,9,10}
```

```
2623 In> Add(testList)
2624 Result> 55
```

2625 **11.19.7 Factorize()**

```
Factorize(list)
```

2626 This function has two calling formats, only one of which is discussed here.

2627 **Factorize(list)** multiplies all the expressions in a list together and returns their  
2628 product:

```
2629 In> Factorize({1,2,3})
```

```
2630 Result> 6
```

2631 **11.20 User Defined Functions**

2632 In computer programming, a **function** is a named sections of code that can be  
2633 **called** from other sections of code. **Values** can be sent to a function for  
2634 processing as part of the **call** and a function always returns a value as its result.

2635 The values that are sent to a function when it is called are called **arguments** and  
2636 a function can accept 0 or more of them. These arguments are placed within  
2637 parentheses.

2638 MathPiper has many predefined functions (some of which have been discussed in  
2639 previous sections) but users can create their own functions too. The following  
2640 program creates a function called **addNums()** which takes two numbers as  
2641 arguments, adds them together, and returns their sum back to the calling code  
2642 as a result:

```
2643 In> addNums(num1,num2) := num1 + num2
```

```
2644 Result> True
```

2645 This line of code defined a new function called **addNums** and specified that it  
2646 will accept two values when it is called. The **first** value will be placed into the  
2647 variable **num1** and the **second** value will be placed into the variable **num2**. The  
2648 code on the **right side** of the assignment operator is then bound to this function  
2649 and it is executed each time the function is called. The following example shows  
2650 the new addNums() function being called multiple times with different values  
2651 being passed to it:

```
2652 In> addNums(2,3)
```

```
2653 Result> 5
```

```
2654 In> addNums(4,5)
```

```
2655 Result> 9
```

```
2656 In> addNums(9,1)
```

2657 `Result> 10`

2658 Notice that, unlike the functions that come with MathPiper, we chose to have this  
2659 function's name start with a **lower case letter**. We could have had addNums()  
2660 begin with an upper case letter but it is a convention in MathPiper for user  
2661 defined function names to begin with a lower case letter to distinguish them  
2662 from the functions that come with MathPiper.

2663 The values that are returned from user defined functions can also be assigned to  
2664 variables. The following example uses a %mathpiper fold to define a function  
2665 called **evenIntegers()** and then this function is used in the MathPiper console:

```
2666 1:%mathpiper
2667 2:
2668 3:evenIntegers(endInteger) :=
2669 4:[
2670 5:   resultList := {};
2671 6:   x := 2;
2672 7:
2673 8:   while(x <= endInteger)
2674 9:   [
2675 10:    resultList := Append(resultList, x);
2676 11:    x := x + 2;
2677 12:   ];
2678 13:
2679 14:   resultList;
2680 15:];
2681 16:
2682 17:%/mathpiper
2683 18:
2684 19:   %output,preserve="false"
2685 20:   Result: True
2686 21: %/output
```

2687 `In> a := evenIntegers(10)`

2688 `Result> {2,4,6,8,10}`

2689 `In> Length(a)`

2690 `Result> 5`

2691 The function evenIntegers() returns a list which contains all the even integers  
2692 from 2 up through the value that was passed into it. The fold was first executed  
2693 in order to define the evenIntegers() function and make it ready for use. The  
2694 evenIntegers() function was then called from the MathPiper console and 10 was  
2695 passed to it. After the function was finished executing, it return a list of even  
2696 integers as a result and this result was assigned to the variable 'a'. We then  
2697 passed the list that was assigned to 'a' to the Length() function in order to  
2698 determine its size.

2699 **11.20.1 Global Variables, Local Variables, & Local()**

2700 The new evenIntegers() function seems to work well, but there is a problem. The  
2701 variables 'x' and resultList were defined inside the function as **global variables**  
2702 which means they are accessible from anywhere, including from within other  
2703 functions, within folds:

```
2704 1:%mathpiper
2705 2:
2706 3:Echo(x, ",", resultList);
2707 4:
2708 5:%/mathpiper
2709 6:
2710 7:    %output,preserve="false"
2711 8:    Result: True
2712 9:
2713 10:    Side effects:
2714 11:    12 ,{2,4,6,8,10}
2715 12:    %/output
```

2716 and from within the MathPiper console:

```
2717 In> x
2718 Result> 12

2719 In> resultList
2720 Result> {2,4,6,8,10}
```

2721 Using global variables inside of functions is usually not a good idea because code  
2722 in other functions and folds might already be using (or will use) the same  
2723 variable names. Global variables which have the same name are the same  
2724 variable. When one section of code changes the value of a given global variable,  
2725 the value is changed everywhere that variable is used and this will eventually  
2726 cause errors.

2727 In order to prevent errors like this, a function named **Local()** can be called  
2728 inside a function to define what are called **local variables**. A **local variable** is  
2729 only accessible inside the function it has been defined in, even if it has the same  
2730 name as a global variable. The following example shows a second version of the  
2731 evenIntegers() function which uses **Local()** to make **x** and **resultList** local  
2732 variables:

```
2733 1:%mathpiper
2734 2:
2735 3:/*
2736 4: This version of evenIntegers() uses Local() to make
2737 5: x and resultList local variables
```

```

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

```

2761 We can verify that x and resultList are now local variables by first clearing them,  
 2762 calling evenIntegers(), and then seeing what x and resultList contain:

```

2763 In> Clear(x, resultList)
2764 Result> True

2765 In> evenIntegers(10)
2766 Result> {2,4,6,8,10}

2767 In> x
2768 Result> x

2769 In> resultList
2770 Result> resultList

```

## 2771 11.21 Applying Functions To List Members

### 2772 11.21.1 Table()

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

2773 The Table() function creates a list of values by doing the following:

- 2774 1) Generating a sequence of values between a "begin\_value" and an  
2775 "end\_value" with each value being incremented by the "step\_amount".
- 2776 2) Placing each value in the sequence into the specified "variable", one value  
2777 at a time.
- 2778 3) Evaluating the defined "expression" (which contains the defined "variable")  
2779 for each value, one at a time.
- 2780 4) Placing the result of each "expression" evaluation into the result list.

2781 This example generates a list which contains the integers 1 through 10:

```
2782 In> Table(x, x, 1, 10, 1)
2783 Result> {1,2,3,4,5,6,7,8,9,10}
```

2784 Notice that the expression in this example is simply the variable itself with no  
2785 other operations performed on it.

2786 The following example is similar to the previous one except that its expression  
2787 multiplies x by 2:

```
2788 In> Table(x*2, x, 1, 10, 1)
2789 Result> {2,4,6,8,10,12,14,16,18,20}
```

2790 Lists which contain decimal values can also be created by setting the  
2791 "step\_amount" to a decimal:

```
2792 In> Table(x, x, 0, 1, .1)
2793 Result> {0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1}
```



2794 **12 THE CONTENT BELOW THIS LINE IS STILL UNDER**  
2795 **DEVELOPMENT**

2796 **12.1 Sets**

2797 The following example shows operations that MathPiper can perform on sets:

```
2798 a = Set([0,1,2,3,4])
2799 b = Set([5,6,7,8,9,0])
2800 a,b
2801 |
2802 ({0, 1, 2, 3, 4}, {0, 5, 6, 7, 8, 9})
```

```
2803 a.cardinality()
2804 |
2805 5
```

```
2806 3 in a
2807 |
2808 True
```

```
2809 3 in b
2810 |
2811 False
```

```
2812 a.union(b)
2813 |
2814 {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

```
2815 a.intersection(b)
2816 |
2817 {0}
```

## 2818 **13 Miscellaneous Topics**

### 2819 **13.1 Errors**

### 2820 **13.2 Style Guide For Expressions**

2821 Always surround the following binary operators with a single space on either  
2822 side: assignment ':=', comparisons (==, <, >, !=, <>, <=, >=, Booleans (and, or,  
2823 not).

2824 Use spaces around the + and – arithmetic operators and no spaces around the  
2825 \*, /, %, and ^ arithmetic operators:

2826  $x = x + 1$

2827  $x = x*3 - 5\%2$

2828  $c = (a + b)/(a - b)$

### 2829 **13.3 Built-in Constants**

2830 MathPiper has a number of mathematical constants built into it and the following  
2831 is a list of some of the more common ones:

2832 Pi, pi: The ratio of the circumference to the diameter of a circle.

2833 E, e: Base of the natural logarithm.

2834 I, i: The imaginary unit quantity.

2835

2836 log2: The natural logarithm of the real number 2.

2837 Infinity, infinity: Can have + or – placed before it to indicate positive or negative  
2838 infinity.

## 2839 **14 Solving Equations**

### 2840 **14.1 Solving Equations Symbolically**

#### 2841 **14.1.1 Symbolic Expressions & Simplify()**

2842 Expressions that contain symbolic variables are called symbolic expressions. In  
2843 the following example,  $b$  is defined to be a symbolic variable and then it is used  
2844 to create the symbolic expression  $2*b$ :

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

2849 As can be seen by this example, the symbolic expression  $2*b$  was placed into an  
2850 object of type SymbolicArithmetic. The expression can also be assigned to a  
2851 variable:

```
2852 m = 2*b
2853 type(m)
2854 |
2855 <class 'sage.calculus.calculus.SymbolicArithmetic'>
```

2856 The following program creates two symbolic expressions, assigns them to  
2857 variables, and then performs operations on them:

```
2858 m = 2*b
2859 n = 3*b
2860 m+n, m-n, m*n, m/n
2861 |
2862 (5*b, -b, 6*b^2, 2/3)
```

2863 Here is another example that multiplies two symbolic expressions together:

```
2864 m = 5 + b
2865 n = 8 + b
2866 y = m*n
2867 y
2868 |
2869 (b + 5)*(b + 8)
```

#### 2870 **14.1.1.1 Expanding And Factoring**

2871 If the expanded form of the expression from the previous section is needed, it is  
2872 easily obtained by calling the `expand()` method (this example assumes the cells in  
2873 the previous section have been run):

```
2874 z = y.expand()
2875 z
2876 |
2877  $b^2 + 13b + 40$ 
2878 The expanded form of the expression has been assigned to variable z and the
2879 factored form can be obtained from z by using the factor() method:
```

```
2880 z.factor()
2881 |
2882  $(b + 5)(b + 8)$ 
2883 By the way, a number can be factored without being assigned to a variable by
2884 placing parentheses around it and calling its factor() method:
```

```
2885 (90).factor()
2886 |
2887  $2 * 3^2 * 5$ 
```

#### 2888 ***14.1.1.2 Miscellaneous Symbolic Expression Examples***

```
2889 var('a,b,c')
2890  $(5a + b + 4c) + (2a + 3b + c)$ 
2891 |
2892  $5c + 4b + 7a$ 
```

```
2893  $(a + b) - (x + 2b)$ 
2894 |
2895  $-x - b + a$ 
```

```
2896  $3a^2 - a(a - 5)$ 
2897 |
2898  $3a^2 - (a - 5)a$ 
```

```
2899  $\_.$ .factor()
2900 |
2901  $a(2a + 5)$ 
```

#### 2902 **14.1.2 Symbolic Equations and The solve() Function**

2903 In addition to working with symbolic expressions, MathPiper is also able to work  
2904 with symbolic equations:

```
2905 var('a')
2906  $\text{type}(x^2 == 16a^2)$ 
2907 |
```

2908 <class 'sage.calculus.equations.SymbolicEquation'>

2909 As can be seen by this example, the symbolic equation  $x^2 == 16a^2$  was  
2910 placed into an object of type SymbolicEquation. A symbolic equation needs to  
2911 use double equals '==' so that it can be assigned to a variable using a single  
2912 equals '=' like this:

2913  $m = x^2 == 16a^2$

2914  $m$ , type( $m$ )

2915 |

2916 ( $x^2 == 16a^2$ , <class 'sage.calculus.equations.SymbolicEquation'>)

2917 Many symbolic equations can be solved algebraically using the solve() function:

2918 solve( $m$ ,  $a$ )

2919 |

2920 [ $a == -x/4$ ,  $a == x/4$ ]

2921 The first parameter in the solve() function accepts a symbolic equation and the  
2922 second parameter accepts the symbolic variable to be solved for.

2923 The solve() function can also solve simultaneous equations:

2924 var('i1,i2,i3,v0')

2925  $a = (i1 - i3)^2 + (i1 - i2)^5 + 10 - 25 == 0$

2926  $b = (i2 - i3)^3 + i2^*1 - 10 + (i2 - i1)^5 == 0$

2927  $c = i3^*14 + (i3 - i2)^3 + (i3 - i1)^2 - (-3^*v0) == 0$

2928  $d = v0 == (i2 - i3)^3$

2929 solve([ $a,b,c,d$ ],  $i1,i2,i3,v0$ )

2930 |

2931 [[ $i1 == 4$ ,  $i2 == 3$ ,  $i3 == -1$ ,  $v0 == 12$ ]]

2932 Notice that, when more than one equation is passed to solve(), they need to be  
2933 placed into a list.

## 2934 **14.2 Solving Equations Numerically**

### 2935 **14.2.1 Roots**

2936 The sqrt() function can be used to obtain the square root of a value, but a more  
2937 general technique is used to obtain other roots of a value. For example, if one  
2938 wanted to obtain the cube root of 8:

2939 8 would be raised to the 1/3 power:

2940  $8^{(1/3)}$

2941 |

2942 2

2943 Due to the order of operations, the rational number 1/3 needs to be placed within  
2944 parentheses in order for it to be evaluated as an exponent.

2945 ***14.3 Finding Roots Graphically And Numerically With The find\_root()***  
2946 ***Method***

2947 Sometimes equations cannot be solved algebraically and the solve() function  
2948 indicates this by returning a copy of the input it was passed. This is shown in the  
2949 following example:

```
2950 f(x) = sin(x) - x - pi/2
2951 eqn = (f == 0)
2952 solve(eqn, x)
2953 |
2954 [x == (2*sin(x) - pi)/2]
```

2955 However, equations that cannot be solved algebraically can be solved both  
2956 graphically and numerically. The following example shows the above equation  
2957 being solved graphically:

```
2958 show(plot(f,-10,10))
2959 |
```

2960 This graph indicates that the root for this equation is a little greater than -2.5.

2961 The following example shows the equation being solved more precisely using the  
2962 find\_root() method:

```
2963 f.find_root(-10,10)
2964 |
2965 -2.309881460010057
```

2966 The -10 and +10 that are passed to the find\_root() method tell it the interval  
2967 within which it should look for roots.

## 2968 **15 Output Forms**

### 2969 ***15.1 LaTeX Is Used To Display Objects In Traditional Mathematics Form***

2970 LaTeX (pronounced lā-tek, <http://en.wikipedia.org/wiki/LaTeX>) is a document  
2971 markup language which is able to work with a wide range of mathematical  
2972 symbols. MathPiper objects will provide LaTeX descriptions of themselves when  
2973 their latex() methods are called. The LaTeX description of an object can also be  
2974 obtained by passing it to the latex() function:

```
2975 a = (2*x^2)/7
```

```
2976 latex(a)
```

```
2977 |
```

```
2978 \frac{{2 \cdot {x}^{{2}} }}{{7}}
```

2979 When this result is fed into LaTeX display software, it will generate traditional  
2980 mathematics form output similar to the following:

2981 The jsMath package which is referenced in is the software that the MathPiper  
2982 Notebook uses to translate LaTeX input into traditional mathematics form  
2983 output.

### 2984 ***15.2 Displaying Mathematical Objects In Traditional Form***

2985 Earlier it was indicated that MathPiper is able to display mathematical objects in  
2986 either text form or traditional form. Up until this point, we have been using text  
2987 form which is the default. If one wants to display a mathematical object in  
2988 traditional form, the show() function can be used. The following example creates  
2989 a mathematical expression and then displays it in both text form and traditional  
2990 form:

```
2991 var('y,b,c')
```

```
2992 z = (3*y^(2*b))/(4*x^c)^2
```

```
2993 #Display the expression in text form.
```

```
2994 z
```

```
2995 |
```

```
2996 3*y^(2*b)/(16*x^(2*c))
```

```
2997 #Display the expression in traditional form.
```

```
2998 show(z)
```

```
2999 |
```

3000 **16 2D Plotting**



3001 **17 High School Math Problems (most of the problems are still in**  
3002 **development)**

3003 **17.1 Pre-Algebra**

3004 Wikipedia entry.

3005 <http://en.wikipedia.org/wiki/Pre-algebra>

3006 (In development...)

3007 **17.1.1 Equations**

3008 Wikipedia entry.

3009 <http://en.wikipedia.org/wiki/Equation>

3010 (In development...)

3011 **17.1.2 Expressions**

3012 Wikipedia entry.

3013 [http://en.wikipedia.org/wiki/Mathematical\\_expression](http://en.wikipedia.org/wiki/Mathematical_expression)

3014 (In development...)

3015 **17.1.3 Geometry**

3016 Wikipedia entry.

3017 <http://en.wikipedia.org/wiki/Geometry>

3018 (In development...)

3019 **17.1.4 Inequalities**

3020 Wikipedia entry.

3021 <http://en.wikipedia.org/wiki/Inequality>

3022 (In development...)

3023 **17.1.5 Linear Functions**

3024 Wikipedia entry.

3025 [http://en.wikipedia.org/wiki/Linear\\_functions](http://en.wikipedia.org/wiki/Linear_functions)

3026 (In development...)

3027 **17.1.6 Measurement**

3028 Wikipedia entry.

3029 <http://en.wikipedia.org/wiki/Measurement>

3030 (In development...)

**3031 17.1.7 Nonlinear Functions**

3032 Wikipedia entry.

3033 [http://en.wikipedia.org/wiki/Nonlinear\\_system](http://en.wikipedia.org/wiki/Nonlinear_system)

3034 (In development...)

**3035 17.1.8 Number Sense And Operations**

3036 Wikipedia entry.

3037 [http://en.wikipedia.org/wiki/Number\\_sense](http://en.wikipedia.org/wiki/Number_sense)

3038 Wikipedia entry.

3039 [http://en.wikipedia.org/wiki/Operation\\_\(mathematics\)](http://en.wikipedia.org/wiki/Operation_(mathematics))

3040 (In development...)

**3041 17.1.8.1 Express an integer fraction in lowest terms**

3042 """

3043 Problem:

3044 Express 90/105 in lowest terms.

3045 Solution:

3046 One way to solve this problem is to factor both the numerator and the  
3047 denominator into prime factors, find the common factors, and then divide both  
3048 the numerator and denominator by these factors.

3049 """

3050 n = 90

3051 d = 105

3052 print n,n.factor()

3053 print d,d.factor()

3054 |

3055 Numerator: 2 \* 3<sup>2</sup> \* 5

3056 Denominator: 3 \* 5 \* 7

3057 """

3058 It can be seen that the factors 3 and 5 each appear once in both the numerator  
3059 and denominator, so we divide both the numerator and denominator by 3\*5:

3060 """

3061 n2 = n/(3\*5)

3062 d2 = d/(3\*5)

3063 print "Numerator2:",n2

3064 print "Denominator2:",d2

3065 |

3066 Numerator2: 6

3067 Denominator2: 7

3068 """

3069 Therefore, 6/7 is 90/105 expressed in lowest terms.

3070 This problem could also have been solved more directly by simply entering  
3071 90/105 into a cell because rational number objects are automatically reduced to  
3072 lowest terms:  
3073 ""  
3074 90/105  
3075 |  
3076 6/7

### 3077 **17.1.9 Polynomial Functions**

3078 Wikipedia entry.  
3079 [http://en.wikipedia.org/wiki/Polynomial\\_function](http://en.wikipedia.org/wiki/Polynomial_function)  
3080 (In development...)

### 3081 **17.2 Algebra**

3082 Wikipedia entry.  
3083 [http://en.wikipedia.org/wiki/Algebra\\_1](http://en.wikipedia.org/wiki/Algebra_1)  
3084 (In development...)

### 3085 **17.2.1 Absolute Value Functions**

3086 Wikipedia entry.  
3087 [http://en.wikipedia.org/wiki/Absolute\\_value](http://en.wikipedia.org/wiki/Absolute_value)  
3088 (In development...)

### 3089 **17.2.2 Complex Numbers**

3090 Wikipedia entry.  
3091 [http://en.wikipedia.org/wiki/Complex\\_numbers](http://en.wikipedia.org/wiki/Complex_numbers)  
3092 (In development...)

### 3093 **17.2.3 Composite Functions**

3094 Wikipedia entry.  
3095 [http://en.wikipedia.org/wiki/Composite\\_function](http://en.wikipedia.org/wiki/Composite_function)  
3096 (In development...)

### 3097 **17.2.4 Conics**

3098 Wikipedia entry.  
3099 <http://en.wikipedia.org/wiki/Conics>  
3100 (In development...)

3101 **17.2.5 Data Analysis**

3102 Wikipedia entry.

3103 [http://en.wikipedia.org/wiki/Data\\_analysis](http://en.wikipedia.org/wiki/Data_analysis)

3104 (In development...)

3105 **17.2.6 Discrete Mathematics**

3106 Wikipedia entry.

3107 [http://en.wikipedia.org/wiki/Discrete\\_mathematics](http://en.wikipedia.org/wiki/Discrete_mathematics)

3108 (In development...)

3109 **17.2.7 Equations**

3110 Wikipedia entry.

3111 <http://en.wikipedia.org/wiki/Equation>

3112 (In development...)

3113 **17.2.7.1 Express a symbolic fraction in lowest terms**

3114 """

3115 Problem:

3116 Express  $(6x^2 - b) / (b - 6ab)$  in lowest terms, where a and b represent  
3117 positive integers.

3118 Solution:

3119 """

3120 `var('a,b')`3121 `n = 6*a^2 - a`3122 `d = b - 6 * a * b`3123 `print n`3124 `print "-----"`3125 `print d`3126 `|`3127 `2`3128 `6 a - a`3129 `-----`3130 `b - 6 a b`

3131 """

3132 We begin by factoring both the numerator and the denominator and then looking  
3133 for common factors:

3134 """

3135 `n2 = n.factor()`3136 `d2 = d.factor()`3137 `print "Factored numerator:",n2.__repr__()`

```

3138 print "Factored denominator:",d2.__repr__()
3139 |
3140 Factored numerator: a*(6*a - 1)
3141 Factored denominator: -(6*a - 1)*b

3142 """
3143 At first, it does not appear that the numerator and denominator contain any
3144 common factors. If the denominator is studied further, however, it can be seen
3145 that if (1 - 6 a) is multiplied by -1,
3146 (6 a - 1) is the result and this factor is also present
3147 in the numerator. Therefore, our next step is to multiply both the numerator and
3148 denominator by -1:
3149 """
3150 n3 = n2 * -1
3151 d3 = d2 * -1
3152 print "Numerator * -1:",n3.__repr__()
3153 print "Denominator * -1:",d3.__repr__()
3154 |
3155 Numerator * -1: -a*(6*a - 1)
3156 Denominator * -1: (6*a - 1)*b

3157 """
3158 Now, both the numerator and denominator can be divided by (6*a - 1) in order to
3159 reduce each to lowest terms:
3160 """
3161 common_factor = 6*a - 1
3162 n4 = n3 / common_factor
3163 d4 = d3 / common_factor
3164 print n4
3165 print "          ---"
3166 print d4
3167 |
3168          - a
3169          ---
3170          b

3171 """
3172 The problem could also have been solved more directly using a
3173 SymbolicArithmetic object:
3174 """
3175 z = n/d
3176 z.simplify_rational()
3177 |
3178 -a/b

```

3179 **17.2.7.2 Determine the product of two symbolic fractions**

3180 Perform the indicated operation:

3181 """

3182 Since symbolic expressions are usually automatically simplified, all that needs to  
3183 be done with this problem is to enter the expression and assign it to a variable:

3184 """

3185 `var('y')`3186 `a = (x/(2*y))^2 * ((4*y^2)/(3*x))^3`3187 `#Display the expression in text form:`3188 `a`3189 `|`3190 `16*y^4/(27*x)`3191 `#Display the expression in traditional form:`3192 `show(a)`3193 `|`3194 **17.2.7.3 Solve a linear equation for x**

3195 Solve

3196 """

3197 Like terms will automatically be combined when this equation is placed into a  
3198 SymbolicEquation object:

3199 """

3200 `a = 5*x + 2*x - 8 == 5*x - 3*x + 7`3201 `a`3202 `|`3203 `7*x - 8 == 2*x + 7`

3204 """

3205 First, lets move the x terms to the left side of the equation by subtracting 2x  
3206 from each side. (Note: remember that the underscore '\_' holds the result of the  
3207 last cell that was executed:

3208 """

3209 `_ - 2*x`3210 `|`3211 `5*x - 8 == 7`

3212 """

3213 Next, add 8 to both sides:

```
3214 """
3215 _+8
3216 |
3217 5*x == 15
3218 """
3219 Finally, divide both sides by 5 to determine the solution:
3220 """
3221 _/5
3222 |
3223 x == 3
3224 """
3225 This problem could also have been solved automatically using the solve()
3226 function:
3227 """
3228 solve(a,x)
3229 |
3230 [x == 3]
```

3231 **17.2.7.4 Solve a linear equation which has fractions**

3232 Solve

```
3233 """
3234 The first step is to place the equation into a SymbolicEquation object. It is good
3235 idea to then display the equation so that you can verify that it was entered
3236 correctly:
3237 """
3238 a = (16*x - 13)/6 == (3*x + 5)/2 - (4 - x)/3
3239 a
3240 |
3241 (16*x - 13)/6 == (3*x + 5)/2 - (4 - x)/3
```

3242 """

3243 In this case, it is difficult to see if this equation has been entered correctly when  
3244 it is displayed in text form so lets also display it in traditional form:

```
3245 """
3246 show(a)
3247 |
```

3248 """

3249 The next step is to determine the least common denominator (LCD) of the  
3250 fractions in this equation so the fractions can be removed:

```
3251 """
3252 lcm([6,2,3])
3253 |
```

3254 6

3255 """

3256 The LCD of this equation is 6 so multiplying it by 6 removes the fractions:

3257 """

3258  $b = a*6$

3259  $b$

3260  $|$

3261  $16*x - 13 == 6*((3*x + 5)/2 - (4 - x)/3)$

3262 """

3263 The right side of this equation is still in factored form so expand it:

3264 """

3265  $c = b.expand()$

3266  $c$

3267  $|$

3268  $16*x - 13 == 11*x + 7$

3269 """

3270 Transpose the 11x to the left side of the equals sign by subtracting 11x from the

3271 SymbolicEquation:

3272 """

3273  $d = c - 11*x$

3274  $d$

3275  $|$

3276  $5*x - 13 == 7$

3277 """

3278 Transpose the -13 to the right side of the equals sign by adding 13 to the

3279 SymbolicEquation:

3280 """

3281  $e = d + 13$

3282  $e$

3283  $|$

3284  $5*x == 20$

3285 """

3286 Finally, dividing the SymbolicEquation by 5 will leave x by itself on the left side  
3287 of the equals sign and produce the solution:

3288 """

3289  $f = e / 5$

3290  $f$

3291  $|$

3292  $x == 4$

3293 """



3294 This problem could have also be solved automatically using the solve() function:  
3295 ""  
3296 solve(a,x)  
3297 |  
3298 [x == 4]

### 3299 **17.2.8 Exponential Functions**

3300 Wikipedia entry.  
3301 [http://en.wikipedia.org/wiki/Exponential\\_function](http://en.wikipedia.org/wiki/Exponential_function)  
3302 (In development...)

### 3303 **17.2.9 Exponents**

3304 Wikipedia entry.  
3305 <http://en.wikipedia.org/wiki/Exponent>  
3306 (In development...)

### 3307 **17.2.10 Expressions**

3308 Wikipedia entry.  
3309 [http://en.wikipedia.org/wiki/Expression\\_\(mathematics\)](http://en.wikipedia.org/wiki/Expression_(mathematics))  
3310 (In development...)

### 3311 **17.2.11 Inequalities**

3312 Wikipedia entry.  
3313 <http://en.wikipedia.org/wiki/Inequality>  
3314 (In development...)

### 3315 **17.2.12 Inverse Functions**

3316 Wikipedia entry.  
3317 [http://en.wikipedia.org/wiki/Inverse\\_function](http://en.wikipedia.org/wiki/Inverse_function)  
3318 (In development...)

### 3319 **17.2.13 Linear Equations And Functions**

3320 Wikipedia entry.  
3321 [http://en.wikipedia.org/wiki/Linear\\_functions](http://en.wikipedia.org/wiki/Linear_functions)  
3322 (In development...)

### 3323 **17.2.14 Linear Programming**

3324 Wikipedia entry.  
3325 [http://en.wikipedia.org/wiki/Linear\\_programming](http://en.wikipedia.org/wiki/Linear_programming)

3326 (In development...)

### 3327 **17.2.15 Logarithmic Functions**

3328 Wikipedia entry.

3329 [http://en.wikipedia.org/wiki/Logarithmic\\_function](http://en.wikipedia.org/wiki/Logarithmic_function)

3330 (In development...)

### 3331 **17.2.16 Logistic Functions**

3332 Wikipedia entry.

3333 [http://en.wikipedia.org/wiki/Logistic\\_function](http://en.wikipedia.org/wiki/Logistic_function)

3334 (In development...)

### 3335 **17.2.17 Matrices**

3336 Wikipedia entry.

3337 [http://en.wikipedia.org/wiki/Matrix\\_\(mathematics\)](http://en.wikipedia.org/wiki/Matrix_(mathematics))

3338 (In development...)

### 3339 **17.2.18 Parametric Equations**

3340 Wikipedia entry.

3341 [http://en.wikipedia.org/wiki/Parametric\\_equation](http://en.wikipedia.org/wiki/Parametric_equation)

3342 (In development...)

### 3343 **17.2.19 Piecewise Functions**

3344 Wikipedia entry.

3345 [http://en.wikipedia.org/wiki/Piecewise\\_function](http://en.wikipedia.org/wiki/Piecewise_function)

3346 (In development...)

### 3347 **17.2.20 Polynomial Functions**

3348 Wikipedia entry.

3349 [http://en.wikipedia.org/wiki/Polynomial\\_function](http://en.wikipedia.org/wiki/Polynomial_function)

3350 (In development...)

### 3351 **17.2.21 Power Functions**

3352 Wikipedia entry.

3353 [http://en.wikipedia.org/wiki/Power\\_function](http://en.wikipedia.org/wiki/Power_function)

3354 (In development...)

### 3355 **17.2.22 Quadratic Functions**

3356 Wikipedia entry.

3357 [http://en.wikipedia.org/wiki/Quadratic\\_function](http://en.wikipedia.org/wiki/Quadratic_function)  
3358 (In development...)

### 3359 **17.2.23 Radical Functions**

3360 Wikipedia entry.  
3361 [http://en.wikipedia.org/wiki/Nth\\_root](http://en.wikipedia.org/wiki/Nth_root)  
3362 (In development...)

### 3363 **17.2.24 Rational Functions**

3364 Wikipedia entry.  
3365 [http://en.wikipedia.org/wiki/Rational\\_function](http://en.wikipedia.org/wiki/Rational_function)  
3366 (In development...)

### 3367 **17.2.25 Sequences**

3368 Wikipedia entry.  
3369 <http://en.wikipedia.org/wiki/Sequence>  
3370 (In development...)

### 3371 **17.2.26 Series**

3372 Wikipedia entry.  
3373 [http://en.wikipedia.org/wiki/Series\\_mathematics](http://en.wikipedia.org/wiki/Series_mathematics)  
3374 (In development...)

### 3375 **17.2.27 Systems of Equations**

3376 Wikipedia entry.  
3377 [http://en.wikipedia.org/wiki/System\\_of\\_equations](http://en.wikipedia.org/wiki/System_of_equations)  
3378 (In development...)

### 3379 **17.2.28 Transformations**

3380 Wikipedia entry.  
3381 [http://en.wikipedia.org/wiki/Transformation\\_\(geometry\)](http://en.wikipedia.org/wiki/Transformation_(geometry))  
3382 (In development...)

### 3383 **17.2.29 Trigonometric Functions**

3384 Wikipedia entry.  
3385 [http://en.wikipedia.org/wiki/Trigonometric\\_function](http://en.wikipedia.org/wiki/Trigonometric_function)  
3386 (In development...)

3387 **17.3 Precalculus And Trigonometry**

3388 Wikipedia entry.

3389 <http://en.wikipedia.org/wiki/Precalculus>

3390 <http://en.wikipedia.org/wiki/Trigonometry>

3391 (In development...)

3392 **17.3.1 Binomial Theorem**

3393 Wikipedia entry.

3394 [http://en.wikipedia.org/wiki/Binomial\\_theorem](http://en.wikipedia.org/wiki/Binomial_theorem)

3395 (In development...)

3396 **17.3.2 Complex Numbers**

3397 Wikipedia entry.

3398 [http://en.wikipedia.org/wiki/Complex\\_numbers](http://en.wikipedia.org/wiki/Complex_numbers)

3399 (In development...)

3400 **17.3.3 Composite Functions**

3401 Wikipedia entry.

3402 [http://en.wikipedia.org/wiki/Composite\\_function](http://en.wikipedia.org/wiki/Composite_function)

3403 (In development...)

3404 **17.3.4 Conics**

3405 Wikipedia entry.

3406 <http://en.wikipedia.org/wiki/Conics>

3407 (In development...)

3408 **17.3.5 Data Analysis**

3409 Wikipedia entry.

3410 [http://en.wikipedia.org/wiki/Data\\_analysis](http://en.wikipedia.org/wiki/Data_analysis)

3411 (In development...)

3412 **17.3.6 Discrete Mathematics**

3413 Wikipedia entry.

3414 [http://en.wikipedia.org/wiki/Discrete\\_mathematics](http://en.wikipedia.org/wiki/Discrete_mathematics)

3415 (In development...)

3416 **17.3.7 Equations**

3417 Wikipedia entry.

3418 <http://en.wikipedia.org/wiki/Equation>

3419 (In development...)

### 3420 **17.3.8 Exponential Functions**

3421 Wikipedia entry.

3422 <http://en.wikipedia.org/wiki/Equation>

3423 (In development...)

### 3424 **17.3.9 Inverse Functions**

3425 Wikipedia entry.

3426 [http://en.wikipedia.org/wiki/Inverse\\_function](http://en.wikipedia.org/wiki/Inverse_function)

3427 (In development...)

### 3428 **17.3.10 Logarithmic Functions**

3429 Wikipedia entry.

3430 [http://en.wikipedia.org/wiki/Logarithmic\\_function](http://en.wikipedia.org/wiki/Logarithmic_function)

3431 (In development...)

### 3432 **17.3.11 Logistic Functions**

3433 Wikipedia entry.

3434 [http://en.wikipedia.org/wiki/Logistic\\_function](http://en.wikipedia.org/wiki/Logistic_function)

3435 (In development...)

### 3436 **17.3.12 Mathematical Analysis**

3437 Wikipedia entry.

3438 [http://en.wikipedia.org/wiki/Mathematical\\_analysis](http://en.wikipedia.org/wiki/Mathematical_analysis)

3439 (In development...)

### 3440 **17.3.13 Matrices And Matrix Algebra**

3441 Wikipedia entry.

3442 [http://en.wikipedia.org/wiki/Matrix\\_\(mathematics\)](http://en.wikipedia.org/wiki/Matrix_(mathematics))

3443 (In development...)

### 3444 **17.3.14 Parametric Equations**

3445 Wikipedia entry.

3446 [http://en.wikipedia.org/wiki/Parametric\\_equation](http://en.wikipedia.org/wiki/Parametric_equation)

3447 (In development...)

**3448 17.3.15 Piecewise Functions**

3449 Wikipedia entry.

3450 [http://en.wikipedia.org/wiki/Piecewise\\_function](http://en.wikipedia.org/wiki/Piecewise_function)

3451 (In development...)

**3452 17.3.16 Polar Equations**

3453 Wikipedia entry.

3454 [http://en.wikipedia.org/wiki/Polar\\_equation](http://en.wikipedia.org/wiki/Polar_equation)

3455 (In development...)

**3456 17.3.17 Polynomial Functions**

3457 Wikipedia entry.

3458 [http://en.wikipedia.org/wiki/Polynomial\\_function](http://en.wikipedia.org/wiki/Polynomial_function)

3459 (In development...)

**3460 17.3.18 Power Functions**

3461 Wikipedia entry.

3462 [http://en.wikipedia.org/wiki/Power\\_function](http://en.wikipedia.org/wiki/Power_function)

3463 (In development...)

**3464 17.3.19 Quadratic Functions**

3465 Wikipedia entry.

3466 [http://en.wikipedia.org/wiki/Quadratic\\_function](http://en.wikipedia.org/wiki/Quadratic_function)

3467 (In development...)

**3468 17.3.20 Radical Functions**

3469 Wikipedia entry.

3470 [http://en.wikipedia.org/wiki/Nth\\_root](http://en.wikipedia.org/wiki/Nth_root)

3471 (In development...)

**3472 17.3.21 Rational Functions**

3473 Wikipedia entry.

3474 [http://en.wikipedia.org/wiki/Rational\\_function](http://en.wikipedia.org/wiki/Rational_function)

3475 (In development...)

**3476 17.3.22 Real Numbers**

3477 Wikipedia entry.

3478 [http://en.wikipedia.org/wiki/Real\\_number](http://en.wikipedia.org/wiki/Real_number)

3479 (In development...)

**3480 17.3.23 Sequences**

3481 Wikipedia entry.

3482 <http://en.wikipedia.org/wiki/Sequence>

3483 (In development...)

**3484 17.3.24 Series**

3485 Wikipedia entry.

3486 [http://en.wikipedia.org/wiki/Series\\_\(mathematics\)](http://en.wikipedia.org/wiki/Series_(mathematics))

3487 (In development...)

**3488 17.3.25 Sets**

3489 Wikipedia entry.

3490 <http://en.wikipedia.org/wiki/Set>

3491 (In development...)

**3492 17.3.26 Systems of Equations**

3493 Wikipedia entry.

3494 [http://en.wikipedia.org/wiki/System\\_of\\_equations](http://en.wikipedia.org/wiki/System_of_equations)

3495 (In development...)

**3496 17.3.27 Transformations**

3497 Wikipedia entry.

3498 [http://en.wikipedia.org/wiki/Transformation\\_\(geometry\)](http://en.wikipedia.org/wiki/Transformation_(geometry))

3499 (In development...)

**3500 17.3.28 Trigonometric Functions**

3501 Wikipedia entry.

3502 [http://en.wikipedia.org/wiki/Trigonometric\\_function](http://en.wikipedia.org/wiki/Trigonometric_function)

3503 (In development...)

**3504 17.3.29 Vectors**

3505 Wikipedia entry.

3506 <http://en.wikipedia.org/wiki/Vector>

3507 (In development...)

**3508 17.4 Calculus**

3509 Wikipedia entry.

3510 <http://en.wikipedia.org/wiki/Calculus>

3511 (In development...)

### 3512 **17.4.1 Derivatives**

3513 Wikipedia entry.

3514 <http://en.wikipedia.org/wiki/Derivative>

3515 (In development...)

### 3516 **17.4.2 Integrals**

3517 Wikipedia entry.

3518 <http://en.wikipedia.org/wiki/Integral>

3519 (In development...)

### 3520 **17.4.3 Limits**

3521 Wikipedia entry.

3522 [http://en.wikipedia.org/wiki/Limit\\_\(mathematics\)](http://en.wikipedia.org/wiki/Limit_(mathematics))

3523 (In development...)

### 3524 **17.4.4 Polynomial Approximations And Series**

3525 Wikipedia entry.

3526 [http://en.wikipedia.org/wiki/Convergent\\_series](http://en.wikipedia.org/wiki/Convergent_series)

3527 (In development...)

## 3528 **17.5 Statistics**

3529 Wikipedia entry.

3530 <http://en.wikipedia.org/wiki/Statistics>

3531 (In development...)

### 3532 **17.5.1 Data Analysis**

3533 Wikipedia entry.

3534 [http://en.wikipedia.org/wiki/Data\\_analysis](http://en.wikipedia.org/wiki/Data_analysis)

3535 (In development...)

### 3536 **17.5.2 Inferential Statistics**

3537 Wikipedia entry.

3538 [http://en.wikipedia.org/wiki/Inferential\\_statistics](http://en.wikipedia.org/wiki/Inferential_statistics)

3539 (In development...)

### 3540 **17.5.3 Normal Distributions**

3541 Wikipedia entry.

3542 [http://en.wikipedia.org/wiki/Normal\\_distribution](http://en.wikipedia.org/wiki/Normal_distribution)

3543 (In development...)



**3544 17.5.4 One Variable Analysis**

3545 Wikipedia entry.

3546 <http://en.wikipedia.org/wiki/Univariate>

3547 (In development...)

**3548 17.5.5 Probability And Simulation**

3549 Wikipedia entry.

3550 <http://en.wikipedia.org/wiki/Probability>

3551 (In development...)

**3552 17.5.6 Two Variable Analysis**

3553 Wikipedia entry.

3554 <http://en.wikipedia.org/wiki/Multivariate>

3555 (In development...)

3556 **18 High School Science Problems**

3557 (In development...)

3558 **18.1 Physics**

3559 Wikipedia entry.

3560 <http://en.wikipedia.org/wiki/Physics>

3561 (In development...)

3562 **18.1.1 Atomic Physics**

3563 Wikipedia entry.

3564 [http://en.wikipedia.org/wiki/Atomic\\_physics](http://en.wikipedia.org/wiki/Atomic_physics)

3565 (In development...)

3566 **18.1.2 Boiling**

3567 Wikipedia entry.

3568 <http://en.wikipedia.org/wiki/Boiling>

3569 (In development...)

3570 **18.1.3 Buoyancy**

3571 Wikipedia entry.

3572 <http://en.wikipedia.org/wiki/Bouyancy>

3573 (In development...)

3574 **18.1.4 Circular Motion**

3575 Wikipedia entry.

3576 [http://en.wikipedia.org/wiki/Circular\\_motion](http://en.wikipedia.org/wiki/Circular_motion)

3577 (In development...)

3578 **18.1.5 Convection**

3579 Wikipedia entry.

3580 <http://en.wikipedia.org/wiki/Convection>

3581 (In development...)

3582 **18.1.6 Density**

3583 Wikipedia entry.

3584 <http://en.wikipedia.org/wiki/Density>

3585 (In development...)

**3586 18.1.7 Diffusion**

3587 Wikipedia entry.

3588 <http://en.wikipedia.org/wiki/Diffusion>

3589 (In development...)

**3590 18.1.8 Dynamics**

3591 Wikipedia entry.

3592 [http://en.wikipedia.org/wiki/Dynamics\\_\(physics\)](http://en.wikipedia.org/wiki/Dynamics_(physics))

3593 (In development...)

**3594 18.1.9 Electricity And Magnetism**

3595 Wikipedia entry.

3596 <http://en.wikipedia.org/wiki/Electricity>

3597 <http://en.wikipedia.org/wiki/Magnetism>

3598 (In development...)

**3599 18.1.10 Energy**

3600 Wikipedia entry.

3601 <http://en.wikipedia.org/wiki/Energy>

3602 (In development...)

**3603 18.1.11 Fluids**

3604 Wikipedia entry.

3605 <http://en.wikipedia.org/wiki/Fluids>

3606 (In development...)

**3607 18.1.12 Freezing**

3608 Wikipedia entry.

3609 <http://en.wikipedia.org/wiki/Freezing>

3610 (In development...)

**3611 18.1.13 Friction**

3612 Wikipedia entry.

3613 <http://en.wikipedia.org/wiki/Friction>

3614 (In development...)

**3615 18.1.14 Heat Transfer**

3616 Wikipedia entry.

3617 [http://en.wikipedia.org/wiki/Heat\\_transfer](http://en.wikipedia.org/wiki/Heat_transfer)  
3618 (In development...)

### 3619 **18.1.15 Insulation**

3620 Wikipedia entry.  
3621 <http://en.wikipedia.org/wiki/Insulation>  
3622 (In development...)

### 3623 **18.1.16 Kinematics**

3624 Wikipedia entry.  
3625 <http://en.wikipedia.org/wiki/Kinematics>  
3626 (In development...)

### 3627 **18.1.17 Light**

3628 Wikipedia entry.  
3629 <http://en.wikipedia.org/wiki/Light>  
3630 (In development...)

### 3631 **18.1.18 Momentum**

3632 Wikipedia entry.  
3633 <http://en.wikipedia.org/wiki/Momentum>  
3634 (In development...)

### 3635 **18.1.19 Newton's Laws**

3636 Wikipedia entry.  
3637 [http://en.wikipedia.org/wiki/Newtons\\_laws](http://en.wikipedia.org/wiki/Newtons_laws)  
3638 (In development...)

### 3639 **18.1.20 Optics**

3640 Wikipedia entry.  
3641 <http://en.wikipedia.org/wiki/Optics>  
3642 (In development...)

### 3643 **18.1.21 Pressure**

3644 Wikipedia entry.  
3645 <http://en.wikipedia.org/wiki/Pressure>  
3646 (In development...)

3647 **18.1.22 Pulleys**

3648 Wikipedia entry.

3649 <http://en.wikipedia.org/wiki/Pulley>

3650 (In development...)

3651 **18.1.23 Relativity**

3652 Wikipedia entry.

3653 <http://en.wikipedia.org/wiki/Relativity>

3654 (In development...)

3655 **18.1.24 Rotational Motion**

3656 Wikipedia entry.

3657 [http://en.wikipedia.org/wiki/Rotational\\_motion](http://en.wikipedia.org/wiki/Rotational_motion)

3658 (In development...)

3659 **18.1.25 Sound**

3660 Wikipedia entry.

3661 <http://en.wikipedia.org/wiki/Sound>

3662 (In development...)

3663 **18.1.26 Thermodynamics**

3664 Wikipedia entry.

3665 <http://en.wikipedia.org/wiki/Thermodynamics>

3666 (In development...)

3667 **18.1.27 Waves**

3668 Wikipedia entry.

3669 <http://en.wikipedia.org/wiki/Waves>

3670 (In development...)

3671 **18.1.28 Work**

3672 Wikipedia entry.

3673 [http://en.wikipedia.org/wiki/Mechanical\\_work](http://en.wikipedia.org/wiki/Mechanical_work)

3674 (In development...)