MathRider For Newbies

by Ted Kosan

Table of Contents

1 Preface

1.1 Dedication

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

1.2 Acknowledgments

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

Susan Addington

Matthew Moelter

1.3 Support Email List

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

2 Introduction

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

2.1 What Is A Super Scientific Calculator?

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

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

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

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

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

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

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

and here is the same formula in text form:

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

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

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

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

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

2.2 What Is MathRider?

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

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

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

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

2.3 What Inspired The Creation Of Mathrider?

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

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

and Steve Yegge's thoughts on learning mathematics:

> 1) Math is a lot easier to pick up after you know how to program. In fact, if you're a halfway decent programmer, you'll find it's almost a snap.
>
> 2) They teach math all wrong in school. Way, WAY wrong. If you teach yourself math the right way, you'll learn faster, remember it longer, and it'll be much more valuable to you as a programmer.
>
> 3) The right way to learn math is breadth-first, not depth-first. You need to survey the space, learn the names of things, figure out what's what.
>
> http://steve-yegge.blogspot.com/2006/03/math-for-programmers.html

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

3 Downloading And Installing MathRider

3.1 Installing Sun's Java Implementation

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

3.1.1 Installing Java On A Windows PC

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

http://java.com/

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

3.1.2 Installing Java On A Macintosh

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

http://developer.apple.com/java.

3.1.3 Installing Java On A Linux PC

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

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

You should also be aware that a number of Linux distributions distribute a non-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 on 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.

3.2.1 Extracting The Archive File For Windows Users

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

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

3.2.2 Extracting The Archive File For Unix Users

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

    tar -xvjf <name of archive file>

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

3.3 MathRider's Directory Structure & Execution Instructions

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

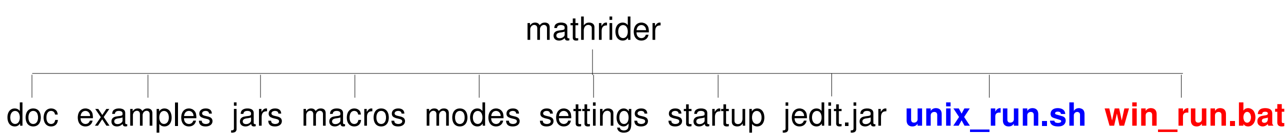

Illustration 1: MathRider's Directory Structure

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

doc - Contains MathRider's documentation files.

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

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

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

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

settings - Contains the application's main settings files.

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

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

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

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

3.3.1 Executing MathRider On Windows Systems

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

3.3.2 Executing MathRider On Unix Systems

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

```
sh unix_run.sh
```

3.3.2.1 MacOS X

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

```
cd   /Applications/mathrider
```

Run mathrider by typing:

```
sh unix_run.sh
```

4 The Graphical User Interface

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

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

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

4.1 Buffers And Text Areas

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

4.2 The Gutter

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

4.3 Menus

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

4.3.1 File

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

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

4.3.2 Edit

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

4.3.3 Search

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

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

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

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

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

4.3.4 Markers

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

4.3.5 Folding

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

Folds are folded and unfolded by pressing on the small black triangles that are next to each fold in the gutter.

4.3.6 View

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

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

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

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

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

4.3.7 Utilities

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

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

4.3.8 Macros

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

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

4.3.9 Plugins

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

4.3.10 Help

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

4.4 The Toolbar

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

5 MathRider's Plugin-Based Extension Mechanism

5.1 What Is A Plugin?

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

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

5.2 Which Plugins Are Currently Included When MathRider Is Installed?

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

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

Calculator - An RPN (Reverse Polish Notation) calculator.

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

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

HotEqn - Renders LaTeX code.

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

LaTeX Tools - Tools to help automate LaTeX editing tasks.

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

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

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

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

5.3 What Kinds Of Plugins Are Possible?

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

5.3.1 Plugins Based On Java Applets

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

5.3.2 Plugins Based On Java Applications

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

5.3.3 Plugins Which Talk To Native Applications

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

6 Exploring The MathRider Application

6.1 The Console

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

6.2 MathPiper Program Files

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

6.3 MathRider Worksheets

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

6.4 Plugins

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

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

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

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

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

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

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

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

7 MathPiper: A Computer Algebra System For Beginners

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

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

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

7.1 Numeric Vs. Symbolic Computations

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

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

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

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

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

```
In> 5-2
Result> 3

In> 3*4
Result> 12

In> 2^3
Result> 8

In> 12/6
Result> 2
```

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

MathPiper can also work with decimal numbers:

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

In> 3.7-2.6
Result> 1.1

In> 2.2*3.9
Result> 8.58

In> 2.2^3
Result> 10.648

In> 9.5/3.2
Result> 9.5/3.2
```

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

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

7.1.1.1 Functions

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

MathPiper has a large number of functions some of which are described in more depth in the MathPiper Documentation section and the MathPiper Programming Fundamentals section. **A complete list of MathPiper's functions can be found in the MathPiperDocs plugin.**

7.1.1.2 Accessing Previous Input And Results

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

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

```
In> 5*8
Result> 40

In> %
Result> 40

In> %*2
Result> 80
```

7.1.1.3 Syntax Errors

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

```
In> 12 \ 6

Error parsing expression, near token \
```

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

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

7.1.2 Using The MathPiper Console As A Symbolic Calculator

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

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

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

Instead of returning a numeric result like 0.8333333333333333333 (which is what a scientific calculator would return) MathPiper added these two rational numbers symbolically and returned $\frac{5}{6}$. If you want to work with this result further, remember that it has also been stored in the **%** symbol:

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

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

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

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

7.1.2.1 Variables

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

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

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

In> a
Result> 5/6

In> Numer(a)
Result> 5

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

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

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

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

In> a
Result> 9
```

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

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

In> a
Result> a
```

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

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

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

In> Factor(14)
Result> 2*7

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

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

```
In> x
Result> x

In> IsBound(x)
Result> False

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

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

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

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

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

8 The MathPiper Documentation Plugin

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

8.1 Function List

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

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

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

8.2 Mini Web Browser Interface

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

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

9 Using MathRider As A Programmer's Text Editor

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

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

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

9.1 Creating, Opening, And Saving Text Files

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

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

9.2 Editing Files

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

9.2.1 Rectangular Selection Mode

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

1) Type 3 or 4 lines of text into a text area.

2) Hold down the **<Alt>** key then slowly press the **backslash key** (****) a few times. The bottom of the MathRider window contains a text field which MathRider uses to communicate information to the user. As **<Alt>** is repeatedly pressed, messages are displayed which read **Rectangular selection is on** and **Rectangular selection is off**.

3) Turn rectangular selection on and then select some text in order to see how this is different than normal selection mode. When you are done experimenting, set rectangular selection mode to **off**.

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.

10 MathRider Worksheet Files

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

10.1 Code Folds

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

```
1:%mathpiper
2:
3:"Hello World!";
4:
5:%/mathpiper
```

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

```
1:%mathpiper
2:
3:"Hello World!";
4:
5:%/mathpiper
6:
7:    %output,preserve="false"
8:      Result: "Hello World!"
9:    %/output
```

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

10.2 Fold Properties

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

```
%mathpiper,output="pretty"

x^2 + x/2 + 3;

%/mathpiper

    %output,preserve="false"
      Result: True

      Side effects:

       2    x
      x  + - + 3
            2
    %/output
```

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

$$x^{2}+\frac{x}{٢}+٣$$

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

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

```
%mathpiper,name="example_1",output="pretty"

x^2 + x/2 + 3;

%/mathpiper
```

```

    %output,preserve="false"
      Result: True

      Side effects:

       2   x
      x  + - + 3
           2
    %/output
```

10.3 Currently Implemented Fold Types And Properties

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

10.3.1 %geogebra & %geogebra_xml.

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

```
%geogebra,clear="true"

//Plot a function.
f(x)=2*sin(x)

//Add a tangent line to the function.
a = 2
(2,0)
t = Tangent[a, f]

%/geogebra

    %output,preserve="false"
      GeoGebra updated.
    %/output
```

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

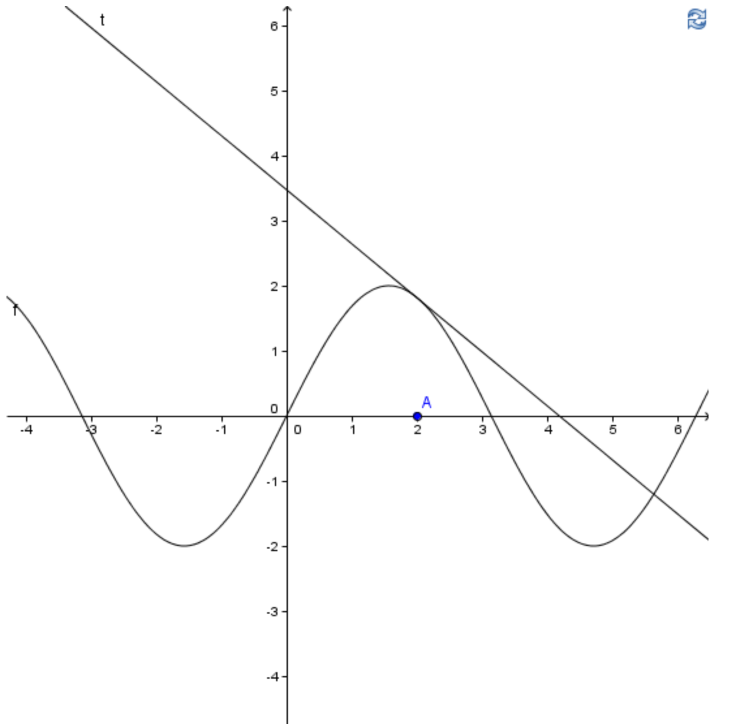

Illustration 2: GeoGebra: sin x and a tangent to it at x=2.

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

```
%geogebra_xml,description="Obtained from .ggb file"

<?xml version="1.0" encoding="utf-8"?>
<geogebra format="3.0">
<gui>
    <show algebraView="true" auxiliaryObjects="true"
    algebraInput="true" cmdList="true"/>
    <splitDivider loc="196" locVertical="400" horizontal="true"/>
    <font  size="12"/>
</gui>
<euclidianView>
    <size  width="540" height="553"/>
    <coordSystem xZero="215.0" yZero="315.0" scale="50.0"
    yscale="50.0"/>
    <evSettings axes="true" grid="true" pointCapturing="3"
    pointStyle="0" rightAngleStyle="1"/>
    <bgColor r="255" g="255" b="255"/>
    <axesColor r="0" g="0" b="0"/>
```

```
    <gridColor r="192" g="192" b="192"/>
    <lineStyle axes="1" grid="10"/>
    <axis id="0" show="true" label="" unitLabel="" tickStyle="1" showNumbers="true"/>
    <axis id="1" show="true" label="" unitLabel="" tickStyle="1" showNumbers="true"/>
    <grid distX="0.5" distY="0.5"/>
</euclidianView>
<kernel>
    <continuous val="true"/>
    <decimals val="2"/>
    <angleUnit val="degree"/>
    <coordStyle val="0"/>
</kernel>
<construction title="" author="" date="">
<expression label ="f" exp="f(x) = 2 sin(x)"/>
<element type="function" label="f">
    <show object="true" label="true"/>
    <objColor r="0" g="0" b="255" alpha="0.0"/>
    <labelMode val="0"/>
    <animation step="0.1"/>
    <fixed val="false"/>
    <breakpoint val="false"/>
    <lineStyle thickness="2" type="0"/>
</element>
<element type="numeric" label="a">
    <value val="2.0"/>
    <show object="false" label="true"/>
    <objColor r="0" g="0" b="0" alpha="0.1"/>
    <labelMode val="1"/>
    <animation step="0.1"/>
    <fixed val="false"/>
    <breakpoint val="false"/>
</element>
<element type="point" label="A">
    <show object="true" label="true"/>
    <objColor r="0" g="0" b="255" alpha="0.0"/>
    <labelMode val="0"/>
    <animation step="0.1"/>
    <fixed val="false"/>
    <breakpoint val="false"/>
    <coords x="2.0" y="0.0" z="1.0"/>
    <coordStyle style="cartesian"/>
    <pointSize val="3"/>
</element>
<command name="Tangent">
    <input a0="a" a1="f"/>
    <output a0="t"/>
</command>
<element type="line" label="t">
```

```
    <show object="true" label="true"/>
    <objColor r="255" g="0" b="0" alpha="0.0"/>
    <labelMode val="0"/>
    <breakpoint val="false"/>
    <coords x="0.8322936730942848" y="1.0" z="-3.4831821998399333"/>
    <lineStyle thickness="2" type="0"/>
    <eqnStyle style="explicit"/>
</element>
</construction>
</geogebra>

%/geogebra_xml

    %output,preserve="false"
      GeoGebra updated.
    %/output
```

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

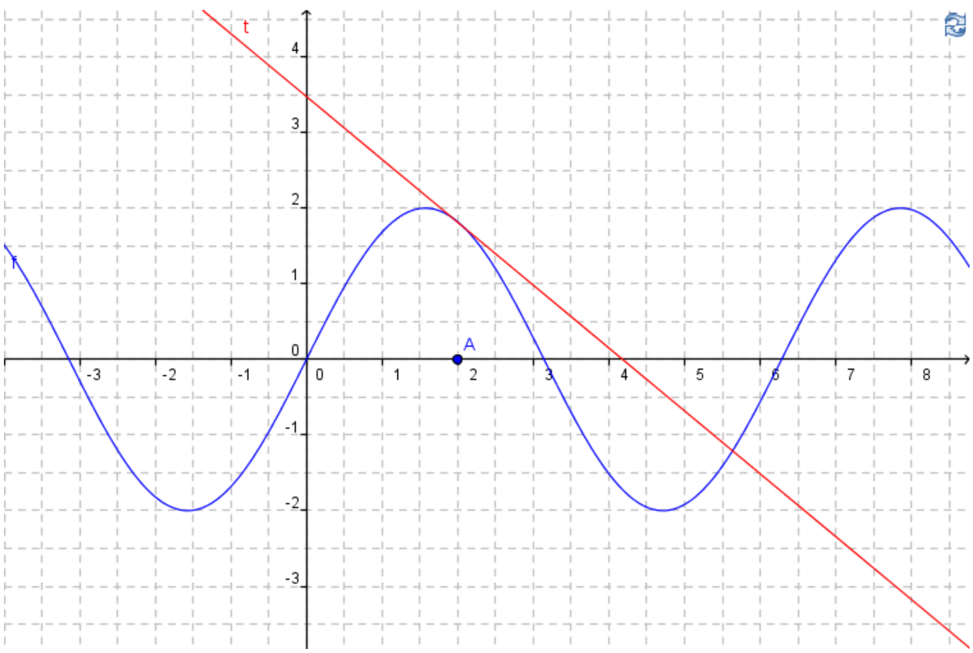

Illustration 3: Generated from %geogebra_xml fold.

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

10.3.2 %hoteqn

Before understanding what the HotEqn (http://www.atp.ruhr-unibochum.de/VCLab/software/HotEqn/HotEqn.html) plugin does, one must first know a little bit about LaTeX. LaTeX is a **markup language** which allows formatting information (such as font size, color, and italics) to be added to plain text. LaTeX was designed for creating technical documents and therefore it is capable of marking up mathematics-related text. The hoteqn plugin accepts input marked up with LaTeX's mathematics-oriented commands and displays it in **traditional mathematics** form. For example, to have HotEqn show $۲^{۳}$, send it 2^{3}:

```
%hoteqn

2^{3}

%/hoteqn

    %output,preserve="false"
      HotEqn updated.
    %/output
```

and it will display:

$$2^{3}$$

To have HotEqn show $۲x^{3}+۱۴x^{2}+\frac{24x}{7}$, send it the following code:

```
%hoteqn

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

%/hoteqn

    %output,preserve="false"
      HotEqn updated.
    %/output
```

and it will display:

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

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

10.3.3 %mathpiper

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

10.3.3.1 Plotting MathPiper Functions With GeoGebra

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

```
%mathpiper,output="geogebra"

x^2;

%/mathpiper
```

Executing this fold will produce the following output:

```
%mathpiper,output="geogebra"

x^2;

%/mathpiper

    %geogebra
      Result: x^2
    %/geogebra
```

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

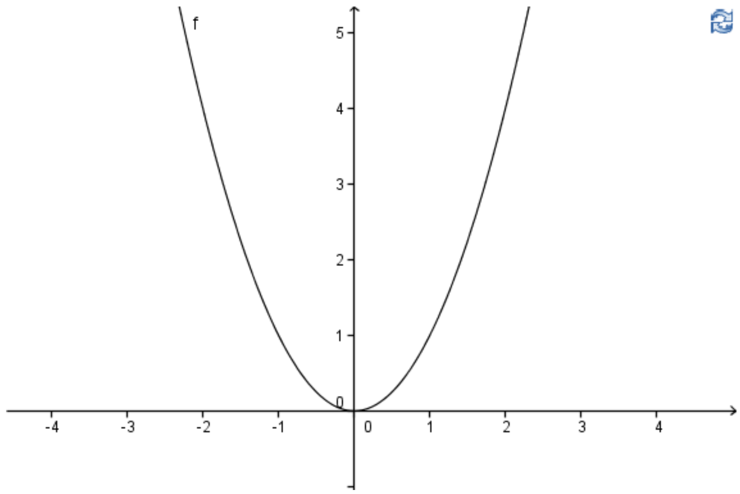

Illustration 4: MathMathPiper Function Plotted With GeoGebra

10.3.3.2 Displaying MathPiper Expressions In Traditional Form With HotEqn

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

```
%mathpiper,output="latex"

((2*x)*(x+3)*(x+4))/9;

%/mathpiper

    %hoteqn
      Result: \frac{2 x \left( x + 3\right)  \left( x + 4\right) }{9}
    %/hoteqn

        %output,preserve="false"
          HotEqn updated.
        %/output
```

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

10.3.4 %output

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

10.3.5 %error

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

10.3.6 %html

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

```
%html,x_size="700",y_size="440"

<html>
    <h1 align="center">HTML Color Values</h1>
    <table border="0" cellpadding="10" cellspacing="1" width="600">
        <tr>
            <th bgcolor="white" colspan="2"></th>
            <th colspan="6">where blue=cc</th>
        </tr>
        <tr>
            <th rowspan="6">where red=</th>
            <th>ff</th>
            <th bgcolor="#ff00cc">ff00cc</th>
            <th bgcolor="#ff33cc">ff33cc</th>
            <th bgcolor="#ff66cc">ff66cc</th>
            <th bgcolor="#ff99cc">ff99cc</th>
            <th bgcolor="#ffcccc">ffcccc</th>
            <th bgcolor="#ffffcc">ffffcc</th>
        </tr>
        <tr>
            <th>cc</th>
            <th bgcolor="#cc00cc">cc00cc</th>
            <th bgcolor="#cc33cc">cc33cc</th>
            <th bgcolor="#cc66cc">cc66cc</th>
            <th bgcolor="#cc99cc">cc99cc</th>
            <th bgcolor="#cccccc">cccccc</th>
            <th bgcolor="#ccffcc">ccffcc</th>
        </tr>
        <tr>
            <th>99</th>
            <th bgcolor="#9900cc">
                <font color="#ffffff">9900cc</font>
```

```
            </th>
            <th bgcolor="#9933cc">9933cc</th>
            <th bgcolor="#9966cc">9966cc</th>
            <th bgcolor="#9999cc">9999cc</th>
            <th bgcolor="#99cccc">99cccc</th>
            <th bgcolor="#99ffcc">99ffcc</th>
        </tr>
        <tr>
            <th>66</th>
            <th bgcolor="#6600cc">
                <font color="#ffffff">6600cc</font>
            </th>
            <th bgcolor="#6633cc">
                <font color="#FFFFFF">6633cc</font>
            </th>
            <th bgcolor="#6666cc">6666cc</th>
            <th bgcolor="#6699cc">6699cc</th>
            <th bgcolor="#66cccc">66cccc</th>
            <th bgcolor="#66ffcc">66ffcc</th>
        </tr>
        <tr>
            <th colspan="1"></th>
            <th>00</th>
            <th>33</th>
            <th>66</th>
            <th>99</th>
            <th>cc</th>
            <th>ff</th>
        </tr>
        <tr>
            <th colspan="2"></th>
            <th colspan="4">where green=</th>
        </tr>
    </table>
</html>

%/html

    %output,preserve="false"

    %/output

```

This code produces the following output:

HTML Color Values

where blue=cc

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

where green=

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

10.3.7 %beanshell

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

10.4 Automatically Inserting Folds & Removing Unpreserved Folds

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

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

11 MathPiper Programming Fundamentals

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

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

11.1 Values and Expressions

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

3

represents the number three, the value:

0.5

represents the number one half, and the value:

"Mathematics is powerful!"

represents an English sentence.

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

3

2 + 3

5 + 6*21/18 - 2^3

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

```
In> 2 + 3
Result> 5
```

11.2 Operators

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

The **subtraction** operator is **−**, the **multiplication** operator is *, **/** is the

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

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

```
In> 5 - 2
Result> 3

In> 3*4
Result> 12

In> 30/3
Result> 10

In> 8%5
Result> 3

In> 2^3
Result> 8
```

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

```
In> -3
Result> -3
```

Subtracting a negative number results in a positive number:

```
In> - -3
Result> 3
```

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

11.3 Operator Precedence

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

^ Exponents are evaluated right to left.

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

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

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

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

And here it is in traditional form:

$$5+6*\frac{21}{18}-\text{٢}^{\text{٣}}$$

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

```
5 + 6*21/18 - 2^3
5 + 6*21/18 - 8
5 + 126/18 - 8
5 + 7 - 8
12 - 8
4
```

Starting with the first expression, MathPiper evaluates the **^** 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**.

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:

```
In> 2 + 4*5
Result> 22
```

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

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

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

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

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

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

11.5 Variables

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

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

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

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

```
In> box
Result> 7
```

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

```
In> box2
Result> box2
```

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

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

```
a := 2
Result> 2

b := 3
Result> 3

a + b
Result> 5

answer := a + b
Result> 5

answer
Result> 5
```

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

11.6 Functions & Function Names

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

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

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

11.7 Functions That Produce Side Effects

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

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

11.7.1 The Echo() and Write() Functions

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

11.7.1.1 Echo()

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

```
In> Echo(1)
Result> True
Side Effects>
1
```

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

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

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

```
Side Effects>
1 3 6
```

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

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

```
%mathpiper

Echo(1);

Echo(1+2);

Echo(2*3);

%/mathpiper

    %output,preserve="false"
      Result: True

      Side effects:
      1
      3
      6
    %/output
```

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

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

```
%mathpiper

a := 1;
b := 2;
c := 3;

%/mathpiper

    %output,preserve="false"
      Result: 3
    %/output
```

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

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

```
%mathpiper

a := 1;
Echo(a);

b := 2;
Echo(b);

c := 3;
Echo(c);

%/mathpiper

    %output,preserve="false"
      Result: True

      Side effects:
      1
      2
      3
    %/output
```

11.7.1.2 Write()

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

```
%mathpiper

Write(1);

Write(1+2);

Echo(2*3);

%/mathpiper

    %output,preserve="false"
      Result: True

```

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

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

11.8 Expressions Are Separated By Semicolons

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

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

```
 1:%mathpiper
 2:
 3:a := 1; Echo(a); b := 2; Echo(b); c := 3; Echo(c);
 4:
 5:%/mathpiper
 6:
 7:    %output,preserve="false"
 8:      Result: True
 9:
10:      Side effects:
11:      1
12:      2
13:      3
14:    %/output
```

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

```
 1:%mathpiper
 2:
 3:a:=1;Echo(a);b:=2;Echo(b);c:= 3;Echo(c);
 4:
 5:%/mathpiper
 6:
 7:    %output,preserve="false"
```

```
      Result: True

      Side effects:
      1
      2
      3
    %/output
```

11.9 Strings

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

```
%mathpiper

a := "Hello, I am a string.";
Echo(a);

%/mathpiper

    %output,preserve="false"
      Result: True

      Side effects:
      Hello, I am a string.
    %/output
```

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

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

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

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

In> a[2]
Result> "e"

In> a[3]
Result> "l"

In> a[4]
Result> "l"

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

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

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

The .. operator is covered in section 11.17.3.1. The .. Range Operator.

11.10 Comments

Source code can often be difficult to understand and therefore all programming languages provide the ability for **comments** to be included in the code. Comments are used to explain what the code near them is doing and they are usually meant to be read by humans instead of being processed by a computer. Comments are ignored when the program is executed.

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

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

x := 2; //Set the variable x equal to 2.


%/mathpiper

    %output,preserve="false"
      Result: 2
```

```
    %/output
```

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

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

```
%mathpiper

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

x := 3;

%/mathpiper

    %output,preserve="false"
      Result: 3
    %/output
```

11.11 Conditional Operators

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

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

Table 2: Conditional Operators

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

```
%mathpiper

// Example 1.
x := 2;
y := 3;

Echo(x, "= ", y, ":", x = y);
Echo(x, "!= ", y, ":", x != y);
Echo(x, "< ", y, ":", x < y);
Echo(x, "<= ", y, ":", x <= y);
Echo(x, "> ", y, ":", x > y);
Echo(x, ">= ", y, ":", x >= y);

%/mathpiper

    %output,preserve="false"
      Result: True

      Side effects:
      2 = 3 :False
      2 != 3 :True
      2 < 3 :True
      2 <= 3 :True
      2 > 3 :False
      2 >= 3 :False
    %/output
```

```
%mathpiper

    // Example 2.
    x := 2;
    y := 2;

    Echo(x, "= ", y, ":", x = y);
    Echo(x, "!= ", y, ":", x != y);
    Echo(x, "< ", y, ":", x < y);
    Echo(x, "<= ", y, ":", x <= y);
    Echo(x, "> ", y, ":", x > y);
    Echo(x, ">= ", y, ":", x >= y);

%/mathpiper

    %output,preserve="false"
      Result: True

      Side effects:
      2 = 2 :True
      2 != 2 :False
      2 < 2 :False
      2 <= 2 :True
      2 > 2 :False
      2 >= 2 :True
    %/output
```

```
%mathpiper

// Example 3.
x := 3;
y := 2;

Echo(x, "= ", y, ":", x = y);
Echo(x, "!= ", y, ":", x != y);
Echo(x, "< ", y, ":", x < y);
Echo(x, "<= ", y, ":", x <= y);
Echo(x, "> ", y, ":", x > y);
Echo(x, ">= ", y, ":", x >= y);

%/mathpiper

    %output,preserve="false"
      Result: True

      Side effects:
      3 = 2 :False
      3 != 2 :True
```

```
      3 < 2 :False
      3 <= 2 :False
      3 > 2 :True
      3 >= 2 :True
    %/output
```

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

() Parentheses are evaluated from the inside out.

^ Then exponents are evaluated right to left.

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

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

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

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

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

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

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

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

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

```
%mathpiper

x := 6;

If(x > 5, Echo(x, "is greater than 5."));

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

```

%/mathpiper

    %output,preserve="false"
      Result: True

      Side effects:
      6 is greater than 5.
      End of program.
    %/output
```

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

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

```
%mathpiper

x := 4;

If(x > 5, Echo(x, "is greater than 5."));

Echo("End of program.");

%/mathpiper

    %output,preserve="false"
      Result: True

      Side effects:
      End of program.
    %/output
```

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

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

```
%mathpiper

x := 4;

If(x > 5,Echo(x,"is greater than 5."),Echo(x,"is NOT greater than 5."));

```

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

%/mathpiper

    %output,preserve="false"
      Result: True

      Side effects:
      4 is NOT greater than 5.
      End of program.
    %/output
```

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

11.13.1 And()

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

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

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

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

In> And(True, False)
Result> False

In> And(False, True)
Result> False

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

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

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

```
expression1 And expression2
```

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

```
In> True And True
Result> True

In> True And False
Result> False

In> False And True
Result> False
```

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

```
%mathpiper

a := 7;
b := 9;

Echo("1: ", a < 5 And b < 10);
Echo("2: ", a > 5 And b > 10);
Echo("3: ", a < 5 And b > 10);
Echo("4: ", a > 5 And b < 10);

If(a > 5 And b < 10, Echo("These expressions are both true."));

%/mathpiper

    %output,preserve="false"
      Result: True

      Side effects:
      1: False
      2: False
      3: False
      4: True
      These expressions are both true.
    %/output
```

11.13.2 Or()

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

Here is the function calling format for Or():

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

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

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

In> Or(False, True)
Result> True

In> Or(False, False)
Result> False

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

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

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

```
expression1 Or expression2
```

and these examples show this notation being used:

```
In> True Or False
Result> True

In> False Or True
Result> True

In> False Or False
Result> False
```

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

```
%mathpiper

a := 7;
```

```
b := 9;

Echo("1: ", a < 5 Or b < 10);
Echo("2: ", a > 5 Or b > 10);
Echo("3: ", a > 5 Or b < 10);
Echo("4: ", a < 5 Or b > 10);

If(a < 5 Or b < 10,Echo("At least one of these expressions is true."));

%/mathpiper

    %output,preserve="false"
      Result: True

      Side effects:
      1: True
      2: True
      3: True
      4: False
      At least one of these expressions is true.
    %/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:

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

In> Not(False)
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:

```
In> Not True
Result> False

In> Not False
Result> True
```

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

```
%mathpiper

Echo("3 = 3 is ", 3 = 3);

Echo("Not 3 = 3 is ", Not 3 = 3);

%/mathpiper

    %output,preserve="false"
      Result: True

      Side effects:
      3 = 3 is True
      Not 3 = 3 is False
    %/output
```

11.14 The While() Looping Function & Bodied Notation

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

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

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

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

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

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

```
%mathpiper

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


/*
    Initialize the variable x to 1
    outside of the While "loop".
*/
x := 1;

While(x <= 10)
[
    Echo(x);

    x := x + 1;  //Increment x by 1.
];

%/mathpiper

    %output,preserve="false"
      Result: True

      Side effects:
      1
      2
      3
      4
      5
      6
      7
      8
      9
      10
    %/output
```

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

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

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

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

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

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

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

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

```
%mathpiper

// Print the integers from 1 to 100.

x := 1;

While(x <= 100)
[
    Write(x);

    x := x + 1;  //Increment x by 1.
];

%/mathpiper

    %output,preserve="false"
      Result: True

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

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

```
%mathpiper

//Print the odd integers from 1 to 99.

x := 1;

While(x <= 100)
[
    Write(x);
    x := x + 2;  //Increment x by 2.
];

%/mathpiper

    %output,preserve="false"
      Result: True

      Side effects:
      1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43
      45 47 49 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83
      85 87 89 91 93 95 97 99
    %/output
```

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

```
%mathpiper

//Print the integers from 1 to 100 in reverse order.

x := 100;

While(x >= 1)
[
    Write(x);
    x := x - 1;  //Decrement x by 1.
];

%/mathpiper

    %output,preserve="false"
      Result: True

      Side effects:
       100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82
       81 80 79 78 77 76 75 74 73 72 71 70 69 68 67 66 65 64 63
       62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44
```

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

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

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

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

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

```
%mathpiper

//Infinite loop example program.

x := 1;
While(x < 10)
[
    answer := x + 1;
];

%/mathpiper

    %output,preserve="false"
      Processing...
    %/output
```

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

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

11.16 Predicate Functions

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

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

In> IsEven(5)
Result> False

In> IsZero(0)
Result> True

In> IsZero(1)
Result> False

In> IsNegativeInteger(-1)
Result> True

In> IsNegativeInteger(1)
Result> False

In> IsPrime(7)
Result> True

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

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

```
In> a
Result> a

In> IsBound(a)
Result> False

In> a := 1
Result> 1

In> IsBound(a)
Result> True

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

```
In> a
Result> a

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

11.17 Lists: Values That Hold Sequences Of Expressions

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

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

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

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

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

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

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

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

In> x[2]
Result> 42

In> x[3]
Result> "Hello"

In> x[4]
Result> 1/2

In> x[5]
```

```
Result> var
```

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

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

In> x[1]
Result> 20

In> x[2]
Result> 30

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

In> x[4]
Result> 40
```

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

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

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

11.17.1 Using While() Loops With Lists

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

```
%mathpiper

//Print each in in the list.

x := {55,93,40,21,7,24,15,14,82};
y := 1;

While(y <= 9)
[
```

```
    Echo(y, "- ", x[y]);
    y := y + 1;
];

%/mathpiper

    %output,preserve="false"
      Result: True

      Side effects:
      1 - 55
      2 - 93
      3 - 40
      4 - 21
      5 - 7
      6 - 24
      7 - 15
      8 - 14
      9 - 82
    %/output
```

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

```
%mathpiper

//Determine if 53 is in the list.

testList := {18,26,32,42,53,43,54,6,97,41};
index := 1;

While(index <= 10)
[
    If(testList[index] = 53,
        Echo("53 was found in the list at position", index));

    index := index + 1;
];

%/mathpiper

    %output,preserve="false"
      Result: True

      Side effects:
      53 was found in the list at position 5
    %/output
```

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

11.17.2 The ForEach() Looping Function

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

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

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

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

```
%mathpiper

//Print all values in a list.

ForEach(x, {50,51,52,53,54,55,56,57,58,59})
[
    Echo(x);
];

%/mathpiper

    %output,preserve="false"
      Result: True

      Side effects:
      50
      51
      52
      53
      54
      55
      56
      57
      58
      59
    %/output
```

11.18 Functions & Operators Which Loop Internally To Process Lists

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

11.18.1 TableForm()

```
TableForm(list)
```

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

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

In> TableForm(testList)
Result> True
Side Effects>
2
4
6
8
10
12
14
16
18
20
```

11.18.2 The .. Range Operator

```
first .. last
```

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

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

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

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

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

11.18.3 Contains()

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

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

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

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

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

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

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

11.18.4 Find()

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

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

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

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

11.18.5 Count()

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

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

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

In> Count(testList, c)
Result> 3

In> Count(testList, e)
Result> 5

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

11.18.6 Select()

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

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

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

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

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

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

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

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

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

11.18.7 The Nth() Function & The [] Operator

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

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

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

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

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

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

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

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

11.18.8 Append() & Nondestructive List Operations

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

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

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

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

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

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

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

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

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

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

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

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

11.18.9 The : Prepend Operator

```
expression : list
```

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

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

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

11.18.10 Concat()

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

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

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

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

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

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

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

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

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

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

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

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

11.18.12 Take()

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

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

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

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

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

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

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

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

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

11.18.13 Drop()

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

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

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

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

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

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

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

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

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

11.18.14 FillList()

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

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

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

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

11.18.15 RemoveDuplicates()

```
RemoveDuplicates(list)
```

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

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

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

11.18.16 Reverse()

```
Reverse(list)
```

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

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

In> Reverse(testList)
Result> {h,g,f,e,d,c,b,a}
```

11.18.17 Partition()

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

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

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

In> Partition(testList, 2)
Result> {{a,b},{c,d},{e,f},{g,h}}
```

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

```
In> Partition(testList, 3)
Result> {{h,b,c},{d,e,f}}
```

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

```
In> Mod(Length(testList), 3)
Result> 2
```

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

11.19 Functions That Work With Integers

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

11.19.1 RandomIntegerVector()

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

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

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

11.19.2 Max() & Min()

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

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

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

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

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

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

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

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

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

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

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

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

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

11.19.3 Div() & Mod()

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

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

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

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

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

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

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

11.19.4 Gcd()

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

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

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

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

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

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

11.19.5 Lcm()

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

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

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

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

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

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

11.19.6 Add()

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

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

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

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

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

In> Add(testList)
Result> 523


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

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

11.19.7 Factorize()

```
Factorize(list)
```

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

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

11.20 User Defined Functions

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

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

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

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

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

```
In> addNums(2,3)
Result> 5

In> addNums(4,5)
Result> 9

In> addNums(9,1)
```

```
Result> 10
```

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

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

```
%mathpiper

evenIntegers(endInteger) :=
[
    resultList := {};
    x := 2;

    While(x <= endInteger)
    [
        resultList := Append(resultList, x);
        x := x + 2;
    ];

    resultList;
];

%/mathpiper

    %output,preserve="false"
      Result: True
    %/output
```

```
In> a := evenIntegers(10)
Result> {2,4,6,8,10}

In> Length(a)
Result> 5
```

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

11.20.1 Global Variables, Local Variables, & Local()

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

```
%mathpiper

Echo(x, ",", resultList);

%/mathpiper

    %output,preserve="false"
      Result: True

      Side effects:
      12 ,{2,4,6,8,10}
    %/output
```

and from within the MathPiper console:

```
In> x
Result> 12

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

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

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

```
%mathpiper

/*
 This version of evenIntegers() uses Local() to make
 x and resultList local variables
```

```
*/

evenIntegers(endInteger) :=
[
    Local(x,resultList);

    resultList := {};
    x := 2;

    While(x <= endInteger)
    [
        resultList := Append(resultList, x);
        x := x + 2;
    ];

    resultList;
];

%/mathpiper

    %output,preserve="false"
      Result: True
    %/output
```

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

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

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

In> x
Result> x

In> resultList
Result> resultList
```

11.21 Applying Functions To List Members

11.21.1 Table()

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

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

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

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

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

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

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

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

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

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

12 THE CONTENT BELOW THIS LINE IS STILL UNDER DEVELOPMENT

12.1 Sets

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

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

a.cardinality()
|
5

3 in a
|
True

3 in b
|
False

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

a.intersection(b)
|
{0}
```

13 Miscellaneous Topics

13.1 Errors

13.2 Style Guide For Expressions

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

Use spaces around the + and − arithmetic operators and no spaces around the * , /, %, and ^ arithmetic operators:

x = x + 1

x = x*3 − 5%2

c = (a + b)/(a − b)

13.3 Built-in Constants

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

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

E, e: Base of the natural logarithm.

I, i: The imaginary unit quantity.

log2: The natural logarithm of the real number 2.

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

14 Solving Equations

14.1 Solving Equations Symbolically

14.1.1 Symbolic Expressions & Simplify()

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

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

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

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

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

```
m = 2*b
n = 3*b
m+n, m-n, m*n, m/n
|
(5*b, -b, 6*b^2, 2/3)
```

Here is another example that multiplies two symbolic expressions together:

```
m = 5 + b
n = 8 + b
y = m*n
y
|
(b + 5)*(b + 8)
```

14.1.1.1 Expanding And Factoring

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

```
z = y.expand()
z
|
b^2 + 13*b + 40
```

The expanded form of the expression has been assigned to variable z and the factored form can be obtained from z by using the factor() method:

```
z.factor()
|
(b + 5)*(b + 8)
```

By the way, a number can be factored without being assigned to a variable by placing parentheses around it and calling its factor() method:

```
(90).factor()
|
2 * 3^2 * 5
```

14.1.1.2 Miscellaneous Symbolic Expression Examples

```
var('a,b,c')

(5*a + b + 4*c) + (2*a + 3*b + c)
|
5*c + 4*b + 7*a

(a + b) - (x + 2*b)
|
-x - b + a

3*a^2 - a*(a -5)
|
3*a^2 - (a - 5)*a

_.factor()
|
a*(2*a + 5)
```

14.1.2 Symbolic Equations and The solve() Function

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

```
var('a')
type(x^2 == 16*a^2)
|
```

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

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

```
m = x^2 == 16*a^2
m, type(m)
|
(x^2 == 16*a^2, <class 'sage.calculus.equations.SymbolicEquation'>)
```

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

```
solve(m, a)
|
[a == -x/4, a == x/4]
```

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

The solve() function can also solve simultaneous equations:

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

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

solve([a,b,c,d], i1,i2,i3,v0)
|
[[i1 == 4, i2 == 3, i3 == -1, v0 == 12]]
```

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

14.2 Solving Equations Numerically

14.2.1 Roots

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

8 would be raised to the 1/3 power:

```
8^(1/3)
|
2
```

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

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

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

```
f(x) = sin(x) - x - pi/2
eqn = (f == 0)
solve(eqn, x)
|
[x == (2*sin(x) - pi)/2]
```

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

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

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

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

```
f.find_root(-10,10)
|
-2.309881460010057
```

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

15 Output Forms

15.1 LaTeX Is Used To Display Objects In Traditional Mathematics Form

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

```
a = (2*x^2)/7
latex(a)
|
\frac{{2 \cdot {x}^{2} }}{7}
```

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

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

15.2 Displaying Mathematical Objects In Traditional Form

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

```
var('y,b,c')
z = (3*y^(2*b))/(4*x^c)^2

#Display the expression in text form.
z
|
3*y^(2*b)/(16*x^(2*c))
#Display the expression in traditional form.
show(z)
|
```

16 2D Plotting

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

17.1 Pre-Algebra

Wikipedia entry.
http://en.wikipedia.org/wiki/Pre-algebra
(In development...)

17.1.1 Equations

Wikipedia entry.
http://en.wikipedia.org/wiki/Equation
(In development...)

17.1.2 Expressions

Wikipedia entry.
http://en.wikipedia.org/wiki/Mathematical_expression
(In development...)

17.1.3 Geometry

Wikipedia entry.
http://en.wikipedia.org/wiki/Geometry
(In development...)

17.1.4 Inequalities

Wikipedia entry.
http://en.wikipedia.org/wiki/Inequality
(In development...)

17.1.5 Linear Functions

Wikipedia entry.
http://en.wikipedia.org/wiki/Linear_functions
(In development...)

17.1.6 Measurement

Wikipedia entry.
http://en.wikipedia.org/wiki/Measurement
(In development...)

17.1.7 Nonlinear Functions

Wikipedia entry.
http://en.wikipedia.org/wiki/Nonlinear_system
(In development...)

17.1.8 Number Sense And Operations

Wikipedia entry.
http://en.wikipedia.org/wiki/Number_sense
Wikipedia entry.
http://en.wikipedia.org/wiki/Operation_(mathematics)
(In development...)

17.1.8.1 Express an integer fraction in lowest terms

```
"""
Problem:
Express 90/105 in lowest terms.

Solution:
One way to solve this problem is to factor both the numerator and the
denominator into prime factors, find the common factors, and then divide both
the numerator and denominator by these factors.
"""
n = 90
d = 105
print n,n.factor()
print d,d.factor()
|
Numerator: 2 * 3^2 * 5
Denominator: 3 * 5 * 7

"""
It can be seen that the factors 3 and 5 each appear once in both the numerator
and denominator, so we divide both the numerator and denominator by 3*5:
"""
n2 = n/(3*5)
d2 = d/(3*5)
print "Numerator2:",n2
print "Denominator2:",d2
|
Numerator2: 6
Denominator2: 7

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

This problem could also have been solved more directly by simply entering 90/105 into a cell because rational number objects are automatically reduced to lowest terms:

```
"""
90/105
|
6/7
```

17.1.9 Polynomial Functions

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

17.2 Algebra

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

17.2.1 Absolute Value Functions

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

17.2.2 Complex Numbers

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

17.2.3 Composite Functions

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

17.2.4 Conics

Wikipedia entry.
http://en.wikipedia.org/wiki/Conics
(In development...)

17.2.5 Data Analysis

Wikipedia entry.
http://en.wikipedia.org/wiki/Data_analysis
(In development...)

17.2.6 Discrete Mathematics

Wikipedia entry.
http://en.wikipedia.org/wiki/Discrete_mathematics
(In development...)

17.2.7 Equations

Wikipedia entry.
http://en.wikipedia.org/wiki/Equation
(In development...)

17.2.7.1 Express a symbolic fraction in lowest terms

```
"""
Problem:
Express (6*x^2 - b) / (b - 6*a*b) in lowest terms, where a and b represent
positive integers.

Solution:
"""

var('a,b')
n = 6*a^2 - a
d = b - 6 * a * b
print n
print "                              ---------"
print d
|
                                 2
                              6 a  - a
                              ---------
                              b - 6 a b

"""
We begin by factoring both the numerator and the denominator and then looking
for common factors:
"""
n2 = n.factor()
d2 = d.factor()
print "Factored numerator:",n2.__repr__()
```

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

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

"""
Now, both the numerator and denominator can be divided by (6*a - 1) in order to
reduce each to lowest terms:
"""
common_factor = 6*a - 1
n4 = n3 / common_factor
d4 = d3 / common_factor
print n4
print "                                   ---"
print d4
|
                                   - a
                                   ---
                                    b

"""
The problem could also have been solved more directly using a
SymbolicArithmetic object:
"""
z = n/d
z.simplify_rational()
|
-a/b
```

17.2.7.2 Determine the product of two symbolic fractions

Perform the indicated operation:

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

var('y')
a = (x/(2*y))^2 * ((4*y^2)/(3*x))^3

#Display the expression in text form:
a
|
16*y^4/(27*x)


#Display the expression in traditional form:
show(a)
|
```

17.2.7.3 Solve a linear equation for x

Solve

```
"""
Like terms will automatically be combined when this equation is placed into a
SymbolicEquation object:
"""
a = 5*x + 2*x - 8 == 5*x - 3*x + 7
a
|
7*x - 8 == 2*x + 7
"""
First, lets move the x terms to the left side of the equation by subtracting 2x
from each side. (Note: remember that the underscore '_' holds the result of the
last cell that was executed:
"""
_ - 2*x
|
5*x - 8 == 7
"""
Next, add 8 to both sides:
```

"""
_+8
|
5*x == 15
"""
Finally, divide both sides by 5 to determine the solution:
"""
_/5
|
x == 3
"""
This problem could also have been solved automatically using the solve() function:
"""
solve(a,x)
|
[x == 3]

17.2.7.4 Solve a linear equation which has fractions

Solve

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

"""
In this case, it is difficult to see if this equation has been entered correctly when it is displayed in text form so lets also display it in traditional form:
"""
show(a)
|

"""
The next step is to determine the least common denominator (LCD) of the fractions in this equation so the fractions can be removed:
"""
lcm([6,2,3])
|

```
6

"""
The LCD of this equation is 6 so multiplying it by 6 removes the fractions:
"""
b = a*6
b
|
16*x - 13 == 6*((3*x + 5)/2 - (4 - x)/3)

"""
The right side of this equation is still in factored form so expand it:
"""
c = b.expand()
c
|
16*x - 13 == 11*x + 7

"""
Transpose the 11x to the left side of the equals sign by subtracting 11x from the
SymbolicEquation:
"""
d = c - 11*x
d
|
5*x - 13 == 7

"""
Transpose the -13 to the right side of the equals sign by adding 13 to the
SymbolicEquation:
"""
e = d + 13
e
|
5*x == 20

"""
Finally, dividing the SymbolicEquation by 5 will leave x by itself on the left side
of the equals sign and produce the solution:
"""
f = e / 5
f
|
x == 4

"""
```

This problem could have also be solved automatically using the solve() function:

```
"""
solve(a,x)
|
[x == 4]
```

17.2.8 Exponential Functions

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

17.2.9 Exponents

Wikipedia entry.
http://en.wikipedia.org/wiki/Exponent
(In development...)

17.2.10 Expressions

Wikipedia entry.
http://en.wikipedia.org/wiki/Expression_(mathematics)
(In development...)

17.2.11 Inequalities

Wikipedia entry.
http://en.wikipedia.org/wiki/Inequality
(In development...)

17.2.12 Inverse Functions

Wikipedia entry.
http://en.wikipedia.org/wiki/Inverse_function
(In development...)

17.2.13 Linear Equations And Functions

Wikipedia entry.
http://en.wikipedia.org/wiki/Linear_functions
(In development...)

17.2.14 Linear Programming

Wikipedia entry.
http://en.wikipedia.org/wiki/Linear_programming

(In development...)

17.2.15 Logarithmic Functions

Wikipedia entry.
http://en.wikipedia.org/wiki/Logarithmic_function
(In development...)

17.2.16 Logistic Functions

Wikipedia entry.
http://en.wikipedia.org/wiki/Logistic_function
(In development...)

17.2.17 Matrices

Wikipedia entry.
http://en.wikipedia.org/wiki/Matrix_(mathematics)
(In development...)

17.2.18 Parametric Equations

Wikipedia entry.
http://en.wikipedia.org/wiki/Parametric_equation
(In development...)

17.2.19 Piecewise Functions

Wikipedia entry.
http://en.wikipedia.org/wiki/Piecewise_function
(In development...)

17.2.20 Polynomial Functions

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

17.2.21 Power Functions

Wikipedia entry.
http://en.wikipedia.org/wiki/Power_function
(In development...)

17.2.22 Quadratic Functions

Wikipedia entry.

http://en.wikipedia.org/wiki/Quadratic_function
(In development...)

17.2.23 Radical Functions

Wikipedia entry.
http://en.wikipedia.org/wiki/Nth_root
(In development...)

17.2.24 Rational Functions

Wikipedia entry.
http://en.wikipedia.org/wiki/Rational_function
(In development...)

17.2.25 Sequences

Wikipedia entry.
http://en.wikipedia.org/wiki/Sequence
(In development...)

17.2.26 Series

Wikipedia entry.
http://en.wikipedia.org/wiki/Series_mathematics
(In development...)

17.2.27 Systems of Equations

Wikipedia entry.
http://en.wikipedia.org/wiki/System_of_equations
(In development...)

17.2.28 Transformations

Wikipedia entry.
http://en.wikipedia.org/wiki/Transformation_(geometry)
(In development...)

17.2.29 Trigonometric Functions

Wikipedia entry.
http://en.wikipedia.org/wiki/Trigonometric_function
(In development...)

17.3 Precalculus And Trigonometry

Wikipedia entry.
http://en.wikipedia.org/wiki/Precalculus

http://en.wikipedia.org/wiki/Trigonometry
(In development...)

17.3.1 Binomial Theorem

Wikipedia entry.
http://en.wikipedia.org/wiki/Binomial_theorem
(In development...)

17.3.2 Complex Numbers

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

17.3.3 Composite Functions

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

17.3.4 Conics

Wikipedia entry.
http://en.wikipedia.org/wiki/Conics
(In development...)

17.3.5 Data Analysis

Wikipedia entry.
http://en.wikipedia.org/wiki/Data_analysis
(In development...)

17.3.6 Discrete Mathematics

Wikipedia entry.
http://en.wikipedia.org/wiki/Discrete_mathematics
(In development...)

17.3.7 Equations

Wikipedia entry.

http://en.wikipedia.org/wiki/Equation
(In development...)

17.3.8 Exponential Functions

Wikipedia entry.
http://en.wikipedia.org/wiki/Equation
(In development...)

17.3.9 Inverse Functions

Wikipedia entry.
http://en.wikipedia.org/wiki/Inverse_function
(In development...)

17.3.10 Logarithmic Functions

Wikipedia entry.
http://en.wikipedia.org/wiki/Logarithmic_function
(In development...)

17.3.11 Logistic Functions

Wikipedia entry.
http://en.wikipedia.org/wiki/Logistic_function
(In development...)

17.3.12 Matrices And Matrix Algebra

Wikipedia entry.
http://en.wikipedia.org/wiki/Matrix_(mathematics)
(In development...)

17.3.13 Mathematical Analysis

Wikipedia entry.
http://en.wikipedia.org/wiki/Mathematical_analysis
(In development...)

17.3.14 Parametric Equations

Wikipedia entry.
http://en.wikipedia.org/wiki/Parametric_equation
(In development...)

17.3.15 Piecewise Functions

Wikipedia entry.
http://en.wikipedia.org/wiki/Piecewise_function
(In development...)

17.3.16 Polar Equations

Wikipedia entry.
http://en.wikipedia.org/wiki/Polar_equation
(In development...)

17.3.17 Polynomial Functions

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

17.3.18 Power Functions

Wikipedia entry.
http://en.wikipedia.org/wiki/Power_function
(In development...)

17.3.19 Quadratic Functions

Wikipedia entry.
http://en.wikipedia.org/wiki/Quadratic_function
(In development...)

17.3.20 Radical Functions

Wikipedia entry.
http://en.wikipedia.org/wiki/Nth_root
(In development...)

17.3.21 Rational Functions

Wikipedia entry.
http://en.wikipedia.org/wiki/Rational_function
(In development...)

17.3.22 Real Numbers

Wikipedia entry.
http://en.wikipedia.org/wiki/Real_number
(In development...)

17.3.23 Sequences

Wikipedia entry.
http://en.wikipedia.org/wiki/Sequence
(In development...)

17.3.24 Series

Wikipedia entry.
http://en.wikipedia.org/wiki/Series_(mathematics)
(In development...)

17.3.25 Sets

Wikipedia entry.
http://en.wikipedia.org/wiki/Set
(In development...)

17.3.26 Systems of Equations

Wikipedia entry.
http://en.wikipedia.org/wiki/System_of_equations
(In development...)

17.3.27 Transformations

Wikipedia entry.
http://en.wikipedia.org/wiki/Transformation_(geometry)
(In development...)

17.3.28 Trigonometric Functions

Wikipedia entry.
http://en.wikipedia.org/wiki/Trigonometric_function
(In development...)

17.3.29 Vectors

Wikipedia entry.
http://en.wikipedia.org/wiki/Vector
(In development...)

17.4 Calculus

Wikipedia entry.
http://en.wikipedia.org/wiki/Calculus
(In development...)

17.4.1 Derivatives

Wikipedia entry.
http://en.wikipedia.org/wiki/Derivative
(In development...)

17.4.2 Integrals

Wikipedia entry.
http://en.wikipedia.org/wiki/Integral
(In development...)

17.4.3 Limits

Wikipedia entry.
http://en.wikipedia.org/wiki/Limit_(mathematics)
(In development...)

17.4.4 Polynomial Approximations And Series

Wikipedia entry.
http://en.wikipedia.org/wiki/Convergent_series
(In development...)

17.5 Statistics

Wikipedia entry.
http://en.wikipedia.org/wiki/Statistics
(In development...)

17.5.1 Data Analysis

Wikipedia entry.
http://en.wikipedia.org/wiki/Data_analysis
(In development...)

17.5.2 Inferential Statistics

Wikipedia entry.
http://en.wikipedia.org/wiki/Inferential_statistics
(In development...)

17.5.3 Normal Distributions

Wikipedia entry.
http://en.wikipedia.org/wiki/Normal_distribution
(In development...)

17.5.4 One Variable Analysis

Wikipedia entry.
http://en.wikipedia.org/wiki/Univariate
(In development...)

17.5.5 Probability And Simulation

Wikipedia entry.
http://en.wikipedia.org/wiki/Probability
(In development...)

17.5.6 Two Variable Analysis

Wikipedia entry.
http://en.wikipedia.org/wiki/Multivariate
(In development...)

18 High School Science Problems

(In development...)

18.1 Physics

Wikipedia entry.
http://en.wikipedia.org/wiki/Physics
(In development...)

18.1.1 Atomic Physics

Wikipedia entry.
http://en.wikipedia.org/wiki/Atomic_physics
(In development...)

18.1.2 Circular Motion

Wikipedia entry.
http://en.wikipedia.org/wiki/Circular_motion
(In development...)

18.1.3 Dynamics

Wikipedia entry.
http://en.wikipedia.org/wiki/Dynamics_(physics)
(In development...)

18.1.4 Electricity And Magnetism

Wikipedia entry.
http://en.wikipedia.org/wiki/Electricity

http://en.wikipedia.org/wiki/Magnetism
(In development...)

18.1.5 Fluids

Wikipedia entry.
http://en.wikipedia.org/wiki/Fluids
(In development...)

18.1.6 Kinematics

Wikipedia entry.
http://en.wikipedia.org/wiki/Kinematics

(In development...)

18.1.7 Light

Wikipedia entry.
http://en.wikipedia.org/wiki/Light
(In development...)

18.1.8 Optics

Wikipedia entry.
http://en.wikipedia.org/wiki/Optics
(In development...)

18.1.9 Relativity

Wikipedia entry.
http://en.wikipedia.org/wiki/Relativity
(In development...)

18.1.10 Rotational Motion

Wikipedia entry.
http://en.wikipedia.org/wiki/Rotational_motion
(In development...)

18.1.11 Sound

Wikipedia entry.
http://en.wikipedia.org/wiki/Sound
(In development...)

18.1.12 Waves

Wikipedia entry.
http://en.wikipedia.org/wiki/Waves
(In development...)

18.1.13 Thermodynamics

Wikipedia entry.
http://en.wikipedia.org/wiki/Thermodynamics
(In development...)

18.1.14 Work

Wikipedia entry.

http://en.wikipedia.org/wiki/Mechanical_work
(In development...)

18.1.15 Energy

Wikipedia entry.
http://en.wikipedia.org/wiki/Energy
(In development...)

18.1.16 Momentum

Wikipedia entry.
http://en.wikipedia.org/wiki/Momentum
(In development...)

18.1.17 Boiling

Wikipedia entry.
http://en.wikipedia.org/wiki/Boiling
(In development...)

18.1.18 Buoyancy

Wikipedia entry.
http://en.wikipedia.org/wiki/Bouyancy
(In development...)

18.1.19 Convection

Wikipedia entry.
http://en.wikipedia.org/wiki/Convection
(In development...)

18.1.20 Density

Wikipedia entry.
http://en.wikipedia.org/wiki/Density
(In development...)

18.1.21 Diffusion

Wikipedia entry.
http://en.wikipedia.org/wiki/Diffusion
(In development...)

18.1.22 Freezing

Wikipedia entry.
http://en.wikipedia.org/wiki/Freezing
(In development...)

18.1.23 Friction

Wikipedia entry.
http://en.wikipedia.org/wiki/Friction
(In development...)

18.1.24 Heat Transfer

Wikipedia entry.
http://en.wikipedia.org/wiki/Heat_transfer
(In development...)

18.1.25 Insulation

Wikipedia entry.
http://en.wikipedia.org/wiki/Insulation
(In development...)

18.1.26 Newton's Laws

Wikipedia entry.
http://en.wikipedia.org/wiki/Newtons_laws
(In development...)

18.1.27 Pressure

Wikipedia entry.
http://en.wikipedia.org/wiki/Pressure
(In development...)

18.1.28 Pulleys

Wikipedia entry.
http://en.wikipedia.org/wiki/Pulley
(In development...)