# 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 And Execution Instructions*

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

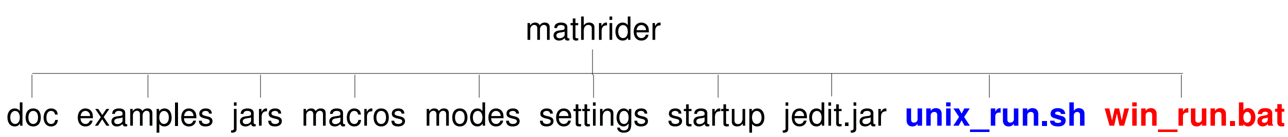


*Illustration 1: MathRider's Directory Structure*

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 $\{n\pi | integer(n)\}$ . 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. 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 0.83333333333333333333 (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 most 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:

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

(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. 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:

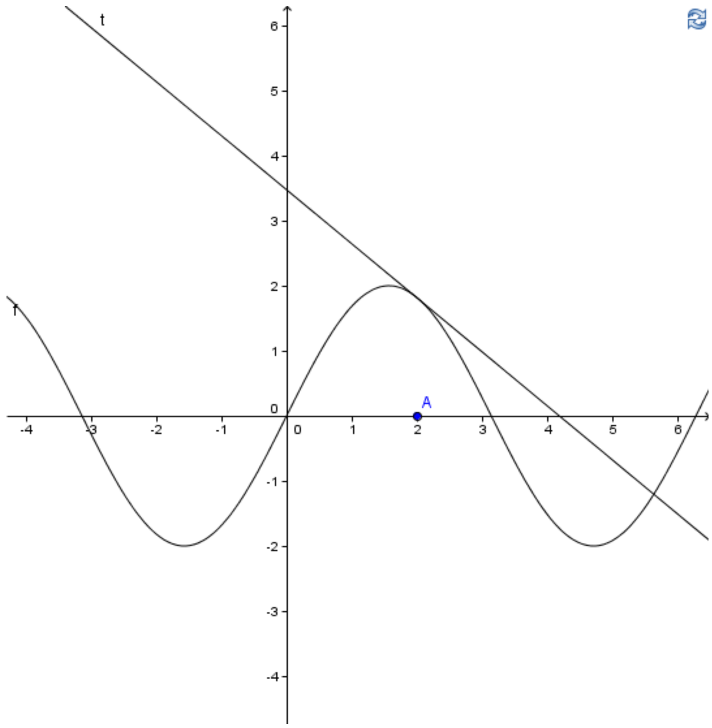


*Illustration 2: GeoGebra: sin x and a tangent to it at x=2.*

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:

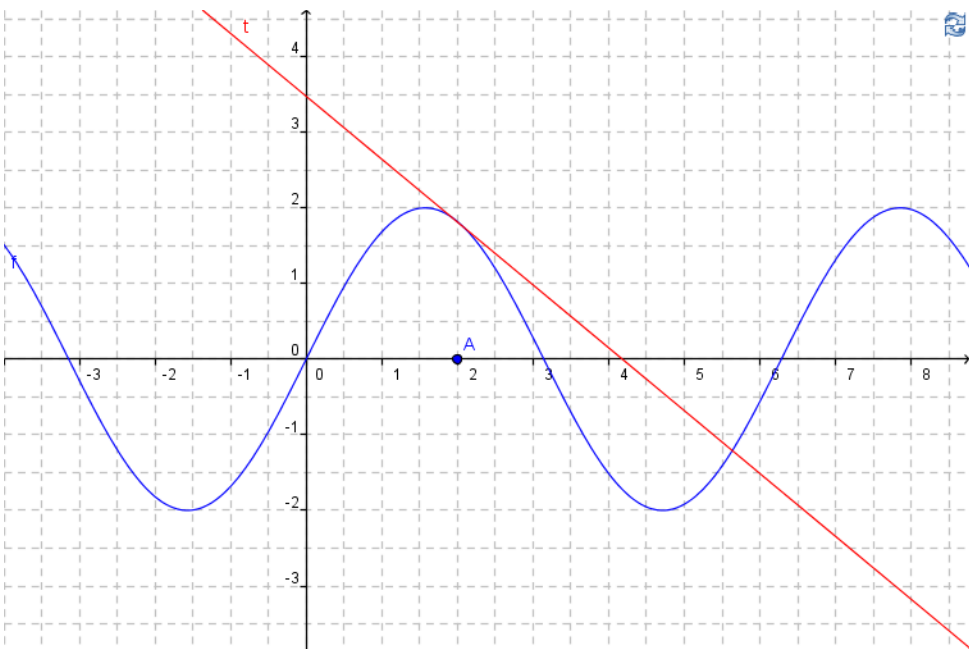


*Illustration 3: Generated from %geogebra_xml fold.*

**%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 ٢٣ , send it 2^{3}:

```
%hoteqn

2^{3}

%/hoteqn

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

and it will display:

$$2^{3}$$

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

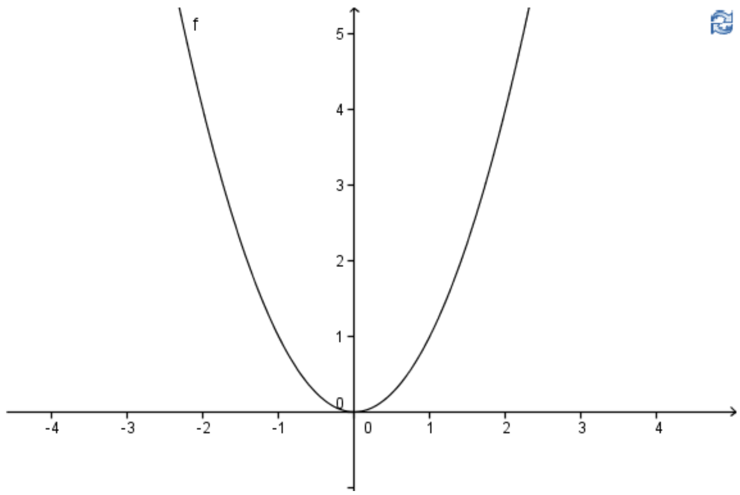


*Illustration 4: MathMathPiper Function Plotted With GeoGebra*

#### 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
```

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

### 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*

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.

# 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:

$$5+6*\frac{21}{\text{١٨}}-2^{3}$$

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

```

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

```
%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
```

## 11.9 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 .. operator:

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

The .. is called the consecutive integer operator and it is covered in section 11.17.3.1. The .. Consecutive Integer 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
```

### 11.17.3 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.

#### *11.17.3.1 The .. Consecutive Integer Operator*

```
first .. last
```

One often needs to create a list of consecutive integers and the **..** consecutive integer 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.

#### *11.17.3.2 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
```

#### 11.17.3.3 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
```

#### 11.17.3.4 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
```

#### 11.17.3.5 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.

#### 11.17.3.6 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
```

#### *11.17.3.7 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 only be used with care by experienced programmers.**

#### *11.17.3.8 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}
```

#### *11.17.3.9 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}
```

#### *11.17.3.10 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}
```

#### 11.17.3.11 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}
```

#### 11.17.3.12 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}
```

#### *11.17.3.13 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}
```

#### *11.17.3.14 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}
```

#### *11.17.3.15 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}
```

#### *11.17.3.16 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.

## *11.18 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.

### 11.18.1 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}
```

### 11.18.2 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
```

### 11.18.3 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
```

### 11.18.4 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
```

### 11.18.5 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
```

### 11.18.6 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
```

### 11.18.7 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
```

# 12 NOTE: THE CONTENT BELOW THIS LINE IS NOT FINISHED YET.

## *12.1 Functions Are Defined Using the def Statement*

The statement that is used to define a function is called def and its syntax specification is as follows:

```
def <function name>(arg1, arg2, ... argN):
    <statement>
    <statement>
    <statement>
    .
    .
    .
```

The def statement contains a header which includes the function's name along with the arguments that can be passed to it. A function can have 0 or more arguments and these arguments are placed within parentheses. The statements that are to be executed when the function is called are placed inside the function using an indented block of code.

The following program defines a function called addnums which takes two numbers as arguments, adds them together, and returns their sum back to the calling code using a return statement:

```
def addnums(num1, num2):
    """
    Returns the sum of num1 and num2.
    """
    answer = num1 + num2
    return answer

#Call the function and have it add 2 to 3.
a = addnums(2, 3)
print a

#Call the function and have it add 4 to 5.
b = addnums(4, 5)
print b
|
5
9
```

The first time this function is called, it is passed the numbers 2 and 3 and these numbers are assigned to the variables num1 and num2 respectively. Argument variables that have objects passed to them during a function call can be used within the function as needed.

Notice that when the function returns back to the caller, the object that was placed to the right of the return statement is made available to the calling code. It is almost as if the function itself is replaced with the object it returns. Another way to think about a returned object is that it is sent out of the left side of the function name in the calling code, through the equals sign, and is assigned to the variable. In the first function call, the object that the function returns is being assigned to the variable 'a' and then this object is printed.

The second function call is similar to the first call, except it passes different numbers (4, 5) to the function.

## *12.2 A Subset Of Functions Included In MathPiper*

MathPiper includes a large number of pre-written functions that can be used for a wide variety of purposes. Table 3 contains a subset of these functions and a longer list of functions can be found in MathPiper's documentation. A more complete list of functions can be found in the MathPiper Reference Manual.

## *12.3 Obtaining Information On MathPiper Functions*

Table 3 includes a list of functions along with a short description of what each one does. This is not enough information, however, to show how to actually use these functions. One way to obtain additional information on any function is to type its name followed by a question mark '?' into a worksheet cell then press the <tab> key:

```
is_even?<tab>
|
File: /opt/sage-2.7.1-debian-32bit-i686-
Linux/local/lib/python2.5/site-packages/sage/misc/functional.py
Type:          <type 'function'>
Definition:     is_even(x)
Docstring:

   Return whether or not an integer x is even, e.g., divisible by 2.

   EXAMPLES:
      sage: is_even(-1)
      False
      sage: is_even(4)
      True
```

```
    sage: is_even(-2)
    True
```

A gray window will then be shown which contains the following information about the function:

File: Gives the name of the file that contains the source code that implements the function. This is useful if you would like to locate the file to see how the function is implemented or to edit it.

Type: Indicates the type of the object that the name passed to the information service refers to.

Definition: Shows how the function is called.

Docstring: Displays the documentation string that has been placed into the source code of this function.

You may obtain help on any of the functions listed in Table 3, or the MathPiper reference manual, using this technique. Also, if you place two question marks '??' after a function name and press the <tab> key, the function's source code will be displayed.

### 12.4 Information Is Also Available On User-Entered Functions

The information service can also be used to obtain information on user-entered functions and a better understanding of how the information service works can be gained by trying this at least once.

If you have not already done so in your current worksheet, type in the addnums function again and execute it:

```
def addnums(num1, num2):
    """
    Returns the sum of num1 and num2.
    """
    answer = num1 + num2
    return answer

#Call the function and have it add 2 to 3.
a = addnums(2, 3)
print a
|
5
```

Then obtain information on this newly-entered function using the technique from the previous section:

```
addnums?<tab>
|
File: /home/sage/sage_notebook/worksheets/root/9/code/8.py
Type: <type 'function'>
Definition: addnums(num1, num2)
Docstring:

    Returns the sum of num1 and num2.
```

This shows that the information that is displayed about a function is obtained from the function's source code.

## 12.5 Examples Which Use Functions Included With MathPiper

The following short programs show how some of the functions listed in Table 3 are used:

```

#Determine the sum of the numbers 1 through 10.
add([1,2,3,4,5,6,7,8,9,10])
|
55

#Cosine of 1 radian.
cos(1.0)
|
0.540302305868140
#Determine the denominator of 15/64.
denominator(15/64)
|
64

#Obtain a list that contains all positive
#integer divisors of 20.
divisors(20)
|
[1, 2, 4, 5, 10, 20]

#Determine the greatest common divisor of 40 and 132.
gcd(40,132)
|
4

#Determine the product of 2, 3, and 4.
mul([2,3,4])
```

```
|
24

#Determine the length of a list.
a = [1,2,3,4,5,6,7]
len(a)
|
7

#Create a list which contains the integers 0 through 10.
a = srange(11)
a
|
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
#Create a list which contains real numbers between
#0.0 and 10.5 in steps of .5.
a = srange(11,step=.5)
a
|
[0.0000000, 0.5000000, 1.000000, 1.500000, 2.000000, 2.500000, 3.000000,
3.500000, 4.000000, 4.500000, 5.000000, 5.500000, 6.000000, 6.500000,
7.000000, 7.500000, 8.000000, 8.500000, 9.000000, 9.500000, 10.00000,
10.50000]
#Create a list which contains the integers -5 through 5.
a = srange(-5,6)
a
|
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5]
#The zip() function takes multiple sequences and groups
#parallel members inside tuples in an output list.  One
#application this is useful for is creating points from
#table data so they can be plotted.
a = [1,2,3,4,5]
b = [6,7,8,9,10]
c = zip(a,b)
c
|
[(1, 6), (2, 7), (3, 8), (4, 9), (5, 10)]
```

### 12.6 Using srange() And zip() With The for Statement

Instead of manually creating a sequence for use by a for statement, srange() can be used to create the sequence automatically:

```
for t in srange(6):
    print t,
```

```
|
0 1 2 3 4 5
```

The for statement can also be used to loop through multiple sequences in parallel using the zip() function:

```
t1 = (0,1,2,3,4)
t2 = (5,6,7,8,9)
for (a,b) in zip(t1,t2):
    print a,b
|
0 5
1 6
2 7
3 8
4 9
```

## *12.7 List Comprehensions*

Up to this point we have seen that if statements, for loops, lists, and functions are each extremely powerful when used individually and together. What is even more powerful, however, is a special statement called a list comprehension which allows them to be used together with a minimum amount of syntax.

Here is the simplified syntax for a list comprehension:

```
[ expression for variable in sequence [if condition] ]
```

What a list comprehension does is to loop through a sequence placing each sequence member into the specified variable in turn. The expression also contains the variable and, as each member is placed into the variable, the expression is evaluated and the result is placed into a new list. When all of the members in the sequence have been processed, the new list is returned.

In the following example, t is the variable, 2*t is the expression, and [1,2,3,4,5] is the sequence:

```
a = [2*t for t in [0,1,2,3,4,5]]
a
|
[0, 2, 4, 6, 8, 10]
```

Instead of manually creating the sequence, the srange() function is often used to create it automatically:

```
a = [2*t for t in srange(6)]
a
```

```
|
[0, 2, 4, 6, 8, 10]
```

An optional if statement can also be used in a list comprehension to filter the results that are placed in the new list:

```
a = [b^2 for b in range(20) if b % 2 == 0]
a
|
[0, 4, 16, 36, 64, 100, 144, 196, 256, 324]
```

In this case, only results that are evenly divisible by 2 are placed in the output list.

# 13 Miscellaneous Topics

## 13.1 Errors

When working on a problem that spans multiple cells in a worksheet, it is often desirable to reference the result of the previous operation. The underscore symbol '_' is used for this purpose as shown in the following example:

```
2 + 3
|
5
_
|
5

_ + 6
|
11

a = _ * 2
a
|
22
```

## 13.2 Style Guide For Expressions

Always surround the following binary operators with a single space on either side: assignment '=', augmented assignment (+=, −=, etc.), comparisons (==, <, >, !=, <>, <=, >=, in, not in, is, is not), 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)
```

Do not use spaces around the equals sign '=' when used to indicate a keyword argument or a default parameter value:

```
a.n(digits=5)
```

## 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.

The following examples show constants being used:

```
a = pi.n()
b = e.n()
c = i.n()
a,b,c
|
(3.14159265358979, 2.71828182845905, 1.00000000000000*I)
```

```
r = 4
a = 2*pi*r
a,a.n()
|
(8*pi, 25.1327412287183)
```

Constants in MathPiper are defined as global variables and a global variable is a variable that is accessible by most MathPiper code, including inside of functions and methods. Since constants are simply variables that have a constant object assigned to them, the variables can be reassigned if needed but then the constant object is lost. If one needs to have a constant reassigned to the variable it is normally associated with, the restore() function can be used. The following program shows how the variable pi can have the object 7 assigned to it and then have its default constant assigned to it again by passing its name inside of quotes to the restore() function:

```
print pi.n()

pi = 7
print pi

restore('pi')
print pi.n()
|
3.14159265358979
7
3.14159265358979
```

If the restore() function is called with no parameters, all reassigned constants are restored to their original values.

### *13.4 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.

### *13.5 Symbolic Variables*

Up to this point, all of the variables we have used have been created during assignment time. For example, in the following code the variable w is created and then the number 8 is assigned to it:

```
w = 7
w
|
7
```

But what if you needed to work with variables that are not assigned to any specific values? The following code attempts to print the value of the variable z, but z has not been assigned a value yet so an exception is returned:

```
print z
|
Exception (click to the left for traceback):
...
NameError: name 'z' is not defined
```

In mathematics, "unassigned variables" are used all the time. Since MathPiper is mathematics oriented software, it has the ability to work with unassigned variables. In MathPiper, unassigned variables are called symbolic variables and they are defined using the var() function. When a worksheet is first opened, the variable x is automatically defined to be a symbolic variable and it will remain so unless it is assigned another value in your code.

The following code was executed on a newly-opened worksheet:

```
print x
type(x)
|
x
<class 'sage.calculus.calculus.SymbolicVariable'>
```

Notice that the variable x has had an object of type SymbolicVariable automatically assigned to it by the MathPiper environment.

If you would like to also use y and z as symbolic variables, the var() function needs to be used to do this. One can either enter var('x,y') or var('x y'). The var() function is designed to accept one or more variable names inside of a string and the names can either be separated by commas or spaces.

The following program shows var() being used to initialize y and z to be symbolic variables:

```
var('y,z')
y,z
|
(y, z)
```

After one or more symbolic variables have been defined, the reset() function can be used to undefine them:

```
reset('y,z')
y,z
|
Exception (click to the left for traceback):
...
NameError: name 'y' is not defined
```

## 13.6 Symbolic Expressions

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)
```

### 13.6.1 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
```

### 13.6.2 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)
```

### 13.6.3 Passing Values To Symbolic Expressions

If values are passed to a symbolic expressions, they will be evaluated and a result will be returned. If the expression only has one variable, then the value can simply be passed to it as follows:

```
a = x^2
a(5)
|
25
```

However, if the expression has two or more variables, each variable needs to have a value assigned to it by name:

```
var('y')
a = x^2 + y
a(x=2, y=3)
|
7
```

## *13.7 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.

### *13.8 Symbolic Mathematical Functions*

MathPiper has the ability to define functions using mathematical syntax. The following example shows a function f being defined that uses x as a variable:

```
f(x) = x^2
f, type(f)
|
(x |--> x^2, <class'sage.calculus.calculus.CallableSymbolicExpression'>)
```

Objects created this way are of type CallableSymbolicExpression which means they can be called as shown in the following example:

```
f(4), f(50), f(.2)
|
(16, 2500, 0.040000000000000010)
```

Here is an example that uses the above CallableSymbolicExpression inside of a loop:

```
a = 0
while a <= 9:
    f(a)
    a = a + 1
|
0
1
4
9
16
25
36
49
64
81
```

The following example accomplishes the same work that the previous example did, except it uses more advanced language features:

```
a = srange(10)
a
|
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
for num in a:
    f(num)
|
0
1
4
9
16
25
36
49
64
81
```

## *13.9 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.

## 13.10 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)
|
```

## 13.11 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.

## 13.12 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}
```

# 14 2D Plotting

## 14.1 The plot() And show() Functions

MathPiper provides a number of ways to generate 2D plots of mathematical functions and one of these ways is to use the plot() function in conjunction with the show() function. The following example shows a symbolic expression being passed to the plot() function as its first parameter. The second parameter indicates where plotting should begin on the X axis and the third parameter indicates where plotting should end:

```
a = x^2
b = plot(a, 0, 10)
type(b)
|
  <class 'sage.plot.plot.Graphics'>
```

Notice that the plot() function does not display the plot. Instead, it creates an object of type sage.plot.plot.Graphics and this object contains the plot data. The show() function can then be used to display the plot:

```
show(b)
|
```

The show() function has 4 parameters called xmin, xmax, ymin, and ymax that can be used to adjust what part of the plot is displayed. It also has a figsize parameter which determines how large the image will be. The following example shows xmin and xmax being used to display the plot between 0 and .05 on the X axis. Notice that the plot() function can be used as the first parameter to the show() function in order to save typing effort (Note: if any other symbolic variable other than x is used, it must first be declared with the var() function):

```
v = 400*e^(-100*x)*sin(200*x)
show(plot(v,0,.1),xmin=0, xmax=.05, figsize=[3,3])
|
```

The ymin and ymax parameters can be used to adjust how much of the y axis is displayed in the above plot:

```
show(plot(v,0,.1),xmin=0, xmax=.05, ymin=0, ymax=100, figsize=[3,3])
|
```

### 14.1.1 Combining Plots And Changing The Plotting Color

Sometimes it is necessary to combine one or more plots into a single plot. The following example combines 6 plots using the show() function:

```
var('t')
p1 = t/4E5
p2 = (5*(t - 8)/2 - 10)/1000000
p3 = (t - 12)/400000
p4 = 0.0000004*(t - 30)
p5 = 0.0000004*(t - 30)
p6 = -0.0000006*(6 - 3*(t - 46)/2)

g1 = plot(p1,0,6,rgbcolor=(0,.2,1))
g2 = plot(p2,6,12,rgbcolor=(1,0,0))
g3 = plot(p3,12,16,rgbcolor=(0,.7,1))
g4 = plot(p4,16,30,rgbcolor=(.3,1,0))
g5 = plot(p5,30,36,rgbcolor=(1,0,1))
g6 = plot(p6,36,50,rgbcolor=(.2,.5,.7))

show(g1+g2+g3+g4+g5+g6,xmin=0, xmax=50, ymin=-.00001, ymax=.00001)
|
```

Notice that the color of each plot can be changed using the rgbcolor parameter. RGB stands for Red, Green, and Blue and the tuple that is assigned to the rgbcolor parameter contains three values between 0 and 1. The first value specifies how much red the plot should have (between 0 and 100%), the second value specifies how much green the plot should have, and the third value specifies how much blue the plot should have.

### 14.1.2 Combining Graphics With A Graphics Object

It is often useful to combine various kinds of graphics into one image. In the following example, 6 points are plotted along with a text label for each plot:

```
"""
Plot the following points on a graph:

A (0,0)
B (9,23)
C (-15,20)
D (22,-12)
E (-5,-12)
F (-22,-4)
"""

#Create a Graphics object which will be used to hold multiple
```

```
# graphics objects.  These graphics objects will be displayed
# on the same image.
g = Graphics()

#Create a list of points and add them to the graphics object.
points=[(0,0), (9,23), (-15,20), (22,-12), (-5,-12), (-22,-4)]
g += point(points)

#Add labels for the points to the graphics object.
for (pnt,letter) in zip(points,['A','B','C','D','E','F']):
    g += text(letter,(pnt[0]-1.5, pnt[1]-1.5))

#Display the combined graphics objects.
show(g,figsize=[5,4])
|
```

First, an empty Graphics object is instantiated and a list of plotted points are created using the point() function. These plotted points are then added to the Graphics object using the += operator. Next, a label for each point is added to the Graphics object using a for loop. Finally, the Graphics object is displayed in the worksheet using the show() function.

Even after being displayed, the Graphics object still contains all of the graphics that have been placed into it and more graphics can be added to it as needed. For example, if a line needed to be drawn between points C and D, the following code can be executed in a separate cell to accomplish this:

```
g += line([(-15,20), (22,-12)])
show(g)
|
```

## *14.2 Advanced Plotting With matplotlib*

MathPiper uses the matplotlib (http://matplotlib.sourceforge.net) library for its plotting needs and if one requires more control over plotting than the plot() function provides, the capabilities of matplotlib can be used directly. While a complete explanation of how matplotlib works is beyond the scope of this book, this section provides examples that should help you to begin using it.

### 14.2.1 Plotting Data From Lists With Grid Lines And Axes Labels

```
x = [1921, 1923, 1925, 1927, 1929, 1931, 1933]
y = [ .05, .6, 4.0, 7.0, 12.0, 15.5, 18.5]
```

```
from matplotlib.backends.backend_agg import FigureCanvasAgg as \
FigureCanvas
from matplotlib.figure import Figure
from matplotlib.ticker import *
fig = Figure()
canvas = FigureCanvas(fig)
ax = fig.add_subplot(111)
ax.xaxis.set_major_formatter( FormatStrFormatter( '%d' ))
ax.yaxis.set_major_locator( MaxNLocator(10) )
ax.yaxis.set_major_formatter( FormatStrFormatter( '%d' ))
ax.yaxis.grid(True, linestyle='-', which='minor')
ax.grid(True, linestyle='-', linewidth=.5)
ax.set_title('US Radios Percentage Gains')
ax.set_xlabel('Year')
ax.set_ylabel('Radios')
ax.plot(x,y, 'go-', linewidth=1.0 )
canvas.print_figure('ex1_linear.png')
|
```

### 14.2.2 Plotting With A Logarithmic Y Axis

```
x = [1926, 1927, 1928, 1929, 1930, 1931, 1932, 1933]
y = [ 4.61,5.24, 10.47, 20.24, 28.83, 43.40, 48.34, 50.80]

from matplotlib.backends.backend_agg import FigureCanvasAgg as \
FigureCanvas
from matplotlib.figure import Figure
from matplotlib.ticker import *
fig = Figure()
canvas = FigureCanvas(fig)
ax = fig.add_subplot(111)
ax.xaxis.set_major_formatter( FormatStrFormatter( '%d' ))
ax.yaxis.set_major_locator( MaxNLocator(10) )
ax.yaxis.set_major_formatter( FormatStrFormatter( '%d' ))
ax.yaxis.grid(True, linestyle='-', which='minor')
ax.grid(True, linestyle='-', linewidth=.5)
ax.set_title('Distance in millions of miles flown by transport airplanes in the US')
ax.set_xlabel('Year')
ax.set_ylabel('Distance')
ax.semilogy(x,y, 'go-', linewidth=1.0 )
canvas.print_figure('ex2_log.png')
|
```

### 14.2.3 Two Plots With Labels Inside Of The Plot

```
x = [20,30,40,50,60,70,80,90,100]
y = [3690,2830,2130,1575,1150,875,735,686,650]
z = [120,680,1860,3510,4780,5590,6060,6340,6520]

from matplotlib.backends.backend_agg import FigureCanvasAgg as \
FigureCanvas
from matplotlib.figure import Figure
from matplotlib.ticker import *
from matplotlib.dates import *
fig = Figure()
canvas = FigureCanvas(fig)
ax = fig.add_subplot(111)
ax.xaxis.set_major_formatter( FormatStrFormatter( '%d' ))
ax.yaxis.set_major_locator( MaxNLocator(10) )
ax.yaxis.set_major_formatter( FormatStrFormatter( '%d' ))
ax.yaxis.grid(True, linestyle='-', which='minor')
ax.grid(True, linestyle='-', linewidth=.5)
ax.set_title('Number of trees vs. total volume of wood')
ax.set_xlabel('Age')
ax.set_ylabel('')
ax.semilogy(x,y, 'bo-', linewidth=1.0 )
ax.semilogy(x,z, 'go-', linewidth=1.0 )
ax.annotate('N', xy=(550, 248),  xycoords='figure pixels')
ax.annotate('V', xy=(180, 230),  xycoords='figure pixels')
canvas.print_figure('ex5_log.png')
|
```

# 15 MathPiper Usage Styles

MathPiper is an extremely flexible environment and therefore there are multiple ways to use it. In this chapter, two MathPiper usage styles are discussed and they are called the Speed style and the OpenOffice Presentation style.

The Speed usage style is designed to solve problems as quickly as possible by minimizing the amount of effort that is devoted to making results look good. This style has been found to be especially useful for solving end of chapter problems that are usually present in mathematics related textbooks.

The OpenOffice Presentation style is designed to allow a person with no mathematical document creation skills to develop mathematical documents with minimal effort. This presentation style is useful for creating homework submissions, reports, articles, books, etc. and this book was developed using this style.

## *15.1 The Speed Usage Style*

(In development...)

## *15.2 The OpenOffice Presentation Usage Style*

(In development...)

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

## 16.1 Pre-Algebra

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

### 16.1.1 Equations

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

### 16.1.2 Expressions

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

### 16.1.3 Geometry

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

### 16.1.4 Inequalities

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

### 16.1.5 Linear Functions

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

### 16.1.6 Measurement

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

### 16.1.7 Nonlinear Functions

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

### 16.1.8 Number Sense And Operations

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

#### *16.1.8.1 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
```

### 16.1.9 Polynomial Functions

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

## *16.2 Algebra*

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

### 16.2.1 Absolute Value Functions

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

### 16.2.2 Complex Numbers

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

### 16.2.3 Composite Functions

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

### 16.2.4 Conics

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

### 16.2.5 Data Analysis

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

### 16.2.6 Discrete Mathematics

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

### 16.2.7 Equations

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

#### *16.2.7.1 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
```

#### 16.2.7.2 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)
|
```

#### 16.2.7.3 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]
```

#### *16.2.7.4 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]
```

### 16.2.8 Exponential Functions

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

### 16.2.9 Exponents

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

### 16.2.10 Expressions

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

### 16.2.11 Inequalities

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

### 16.2.12 Inverse Functions

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

### 16.2.13 Linear Equations And Functions

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

### 16.2.14 Linear Programming

Wikipedia entry.

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

### 16.2.15 Logarithmic Functions

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

### 16.2.16 Logistic Functions

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

### 16.2.17 Matrices

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

### 16.2.18 Parametric Equations

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

### 16.2.19 Piecewise Functions

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

### 16.2.20 Polynomial Functions

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

### 16.2.21 Power Functions

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

### 16.2.22 Quadratic Functions

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

### 16.2.23 Radical Functions

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

### 16.2.24 Rational Functions

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

### 16.2.25 Sequences

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

### 16.2.26 Series

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

### 16.2.27 Systems of Equations

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

### 16.2.28 Transformations

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

### 16.2.29 Trigonometric Functions

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

## *16.3 Precalculus And Trigonometry*

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

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

### 16.3.1 Binomial Theorem

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

### 16.3.2 Complex Numbers

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

### 16.3.3 Composite Functions

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

### 16.3.4 Conics

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

### 16.3.5 Data Analysis

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

### 16.3.6 Discrete Mathematics

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

### 16.3.7 Equations

Wikipedia entry.

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

### 16.3.8 Exponential Functions

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

### 16.3.9 Inverse Functions

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

### 16.3.10 Logarithmic Functions

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

### 16.3.11 Logistic Functions

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

### 16.3.12 Matrices And Matrix Algebra

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

### 16.3.13 Mathematical Analysis

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

### 16.3.14 Parametric Equations

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

### 16.3.15 Piecewise Functions

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

### 16.3.16 Polar Equations

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

### 16.3.17 Polynomial Functions

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

### 16.3.18 Power Functions

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

### 16.3.19 Quadratic Functions

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

### 16.3.20 Radical Functions

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

### 16.3.21 Rational Functions

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

### 16.3.22 Real Numbers

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

### 16.3.23 Sequences

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

### 16.3.24 Series

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

### 16.3.25 Sets

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

### 16.3.26 Systems of Equations

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

### 16.3.27 Transformations

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

### 16.3.28 Trigonometric Functions

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

### 16.3.29 Vectors

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

## *16.4 Calculus*

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

### 16.4.1 Derivatives

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

### 16.4.2 Integrals

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

### 16.4.3 Limits

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

### 16.4.4 Polynomial Approximations And Series

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

## *16.5 Statistics*

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

### 16.5.1 Data Analysis

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

### 16.5.2 Inferential Statistics

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

### 16.5.3 Normal Distributions

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

### 16.5.4 One Variable Analysis

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

### 16.5.5 Probability And Simulation

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

### 16.5.6 Two Variable Analysis

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

# 17 High School Science Problems

(In development...)

## *17.1 Physics*

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

### 17.1.1 Atomic Physics

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

### 17.1.2 Circular Motion

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

### 17.1.3 Dynamics

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

### 17.1.4 Electricity And Magnetism

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

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

### 17.1.5 Fluids

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

### 17.1.6 Kinematics

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

(In development...)

### 17.1.7 Light

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

### 17.1.8 Optics

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

### 17.1.9 Relativity

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

### 17.1.10 Rotational Motion

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

### 17.1.11 Sound

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

### 17.1.12 Waves

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

### 17.1.13 Thermodynamics

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

### 17.1.14 Work

Wikipedia entry.

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

### 17.1.15 Energy

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

### 17.1.16 Momentum

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

### 17.1.17 Boiling

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

### 17.1.18 Buoyancy

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

### 17.1.19 Convection

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

### 17.1.20 Density

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

### 17.1.21 Diffusion

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

### 17.1.22 Freezing

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

### 17.1.23 Friction

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

### 17.1.24 Heat Transfer

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

### 17.1.25 Insulation

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

### 17.1.26 Newton's Laws

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

### 17.1.27 Pressure

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

### 17.1.28 Pulleys

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

# 18 Fundamentals Of Computation

## *18.1 What Is A Computer?*

Many people think computers are difficult to understand because they are complex. Computers are indeed complex, but this is not why they are difficult to understand. Computers are difficult to understand because only a small part of a computer exists in the physical world. The physical part of a computer is the only part a human can see and the rest of a computer exists in a nonphysical world which is invisible. This invisible world is the world of ideas and most of a computer exists as ideas in this nonphysical world.

The key to understanding computers is to understand that the purpose of these idea-based machines is to automatically manipulate ideas of all types. The name 'computer' is not very helpful for describing what computers really are and perhaps a better name for them would be Idea Manipulation Devices or IMDs.

Since ideas are nonphysical objects, they cannot be brought into the physical world and neither can physical objects be brought into the world of ideas. Since these two worlds are separate from each other, the only way that physical objects can manipulate objects in the world of ideas is through remote control via symbols.

12.2 What Is A Symbol?

A symbol is an object that is used to represent another object. Drawing 5 shows an example of a symbol of a telephone which is used to represent a physical telephone.

The symbol of a telephone shown in Drawing 5 is usually created with ink printed on a flat surface ( like a piece of paper ). In general, though, any type of physical matter ( or property of physical matter ) that is arranged into a pattern can be used as a symbol.

12.3 Computers Use Bit Patterns As Symbols

Symbols which are made of physical matter can represent all types of physical objects, but they can also be used to represent nonphysical objects in the world of ideas. ( see Drawing 6 )

Among the simplest symbols that can be formed out of physical matter are bits and patterns of bits. A single bit can only be placed into two states which are the on state and the off state. When written, typed, or drawn, a bit in the on state is represented by the numeral 1 and when it is in the off state it is represented by the numeral 0. Patterns of bits look like the following when they are written, typed, or drawn: 101, 100101101, 0101001100101, 10010.

Drawing 7 shows how bit patterns can be used just as easily as any other symbols made of physical matter to represent nonphysical ideas.

Other methods for forming physical matter into bits and bit patterns include: varying the tone of an audio signal between two frequencies, turning a light on and off, placing or removing a magnetic field on the surface of an object, and changing the voltage level between two levels in an electronic device. Most computers use the last method to hold bit patterns that represent ideas.

A computer's internal memory consists of numerous "boxes" called memory locations and each memory location contains a bit pattern that can be used to represent an idea. Most computers contain millions of memory locations which allow them to easily reference millions of ideas at the same time. Larger computers contain billions of memory locations. For example, a typical personal computer purchased in 2007 contains over 1 billion memory locations.

Drawing 8 shows a section of the internal memory of a small computer along with the bit patterns that this memory contains.

Each of the millions of bit pattern symbols in a computer's internal memory are capable of representing any idea a human can think of. The large number of bit patterns that most computers contain, however, would be difficult to keep track of without the use of some kind of organizing system.

The system that computers use to keep track of the many bit patterns they contain consists of giving each memory location a unique address as shown in Drawing 9.

## *18.2 Contextual Meaning*

At this point you may be wondering "how one can determine what the bit patterns in a memory location, or a set of memory locations, mean?" The answer to this question is that a concept called contextual meaning gives bit patterns their meaning.

Context is the circumstances within which an event happens or the environment within which something is placed. Contextual meaning, therefore, is the meaning that a context gives to the events or things that are placed within it.

Most people use contextual meaning every day, but they are not aware of it. Contextual meaning is a very powerful concept and it is what enables a computer's memory locations to reference any idea that a human can think of. Each memory location can hold a bit pattern, but a human can have that bit pattern mean anything they wish. If more bits are needed to hold a given pattern than are present in a single memory location, the pattern can be spread across more than one location.

## 18.3 Variables

Computers are very good at remembering numbers and this allows them to keep track of numerous addresses with ease. Humans, however, are not nearly as good at remembering numbers as computers are and so a concept called a variable was invented to solve this problem.

A variable is a name that can be associated with a memory address so that humans can refer to bit pattern symbols in memory using a name instead of a number. Drawing 10 shows four variables that have been associated with 4 memory addresses inside of a computer.

The variable names garage_width and garage_length are referencing memory locations that hold patterns that represent the dimensions of a garage and the variable names x and y are referencing memory locations that might represent numbers in an equation. Even though this description of the above variables is accurate, it is fairly tedious to use and therefore most of the time people just say or write something like "the variable garage_length holds the length of the garage."

A variable is used to symbolically represent an attribute of an object. Even though a typical personal computer is capable of holding millions of variables, most objects possess a greater number of attributes than the capacity of most computers can hold. For example, a 1 kilogram rock contains approximately 10,000,000,000,000,000,000,000,000 atoms. [1] Representing even just the positions of this rock's atoms is currently well beyond the capacity of even the most advanced computer. Therefore, computers usually work with models of objects instead of complete representations of them.

## 18.4 Models

A model is a simplified representation of an object that only references some of its attributes. Examples of typical object attributes include weight, height, strength, and color. The attributes that are selected for modeling are chosen for a given purpose. The more attributes that are represented in the model, the more expensive the model is to make. Therefore, only those attributes that are absolutely needed to achieve a given purpose are usually represented in a model. The process of selecting only some of an object's attributes when developing a model of it is called abstraction.

The following is an example which illustrates the process of problem solving using models. Suppose we wanted to build a garage that could hold 2 cars along with a workbench, a set of storage shelves, and a riding lawn mower. Assuming that the garage will have an adequate ceiling height, and that we do not want to build the garage any larger than it needs to be for our stated purpose, how could an adequate length and width be determined for the garage?

One strategy for determining the size of the garage is to build perhaps 10 garages of various sizes in a large field. When the garages are finished, take 2 cars to the field along with a workbench, a set of storage shelves, and a riding lawn mower. Then, place these items into each garage in turn to see which is the smallest one that these items will fit into without being too cramped.

The test garages in the field can then be discarded and a garage which is the same size as the one that was chosen could be built at the desired location. Unfortunately, 11 garages would need to be built using this strategy instead of just one and this would be very expensive and inefficient.

A way to solve this problem less expensively is by using a model of the garage and models of the items that will be placed inside it. Since we only want to determine the dimensions of the garage's floor, we can make a scaled down model of just its floor using a piece of paper.

Each of the items that will be placed into the garage could also be represented by scaled-down pieces of paper. Then, the pieces of paper that represent the items can be placed on top of the the large piece of paper that represents the floor and these smaller pieces of paper can be moved around to see how they fit. If the items are too cramped, a larger piece of paper can be cut to represent the floor and, if the items have too much room, a smaller piece of paper for the floor can be cut.

When a good fit is found, the length and width of the piece of paper that represents the floor can be measured and then these measurements can be scaled up to the units used for the full-size garage. With this method, only a few pieces of paper are needed to solve the problem instead of 10 full-size garages that will later be discarded.

The only attributes of the full-sized objects that were copied to the pieces of paper were the object's length and width. As this example shows, paper models are significantly easier to work with than the objects they represent. However, computer variables are even easier to use for modeling than paper or almost any other kind of modeling mechanism.

At this point, though, the paper-based modeling technique has one important advantage over the computer variables we have look at. The paper model was able to be changed by moving the item models around and changing the size of the paper garage floor. The variables we have discussed so have been given the ability to represent an object attribute, but no mechanism has been given yet that would allow the variable's to change. A computer without the ability to change the contents of its variables would be practically useless.

## *18.5 Machine Language*

Earlier is was stated that bit patterns in a computer's memory locations can be used to represent any ideas that a human can think of. If memory locations can represent any idea, this means that they can reference ideas that represent instructions which tell a computer how to automatically manipulate the variables in its memory.

The part of a computer that follows the instructions that are in its memory is called a Central Processing Unit ( CPU ) or a microprocessor. When a microprocessor is following instructions in its memory, it is also said to be running them or executing them.

Microprocessors are categorized into families and each microprocessor family has its own set of instructions ( called an instruction set ) that is different than the instructions that other microprocessor family's use. A microprocessor's instruction set represents the building blocks of a language that can be used to tell it what to do. This language is formed by placing sequences of instructions from the instruction set into memory and it the only language that a microprocessor is able to understand. Since this is the only language a microprocessor is able to understand, it is called machine language. A sequence of machine language instructions is called a computer program and a person who creates sequences of machine language instructions in order to tell the computer what to do is called a programmer.

We will now look at what the instruction set of a simple microprocessor looks like along with a simple program which has been developed using this instruction set.

Here is the instruction set for the 6500 family of microprocessors:

ADC ADd memory to accumulator with Carry.
AND AND memory with accumulator.
ASL Arithmetic Shift Left one bit.
BCC Branch on Carry Clear.
BCS Branch on Carry Set.
BEQ Branch on result EQual to zero.
BIT test BITs in accumulator with memory.
BMI Branch on result MInus.
BNE Branch on result Not Equal to zero.
BPL Branch on result PLus).
BRK force Break.
BVC Branch on oVerflow flag Clear.

BVS Branch on oVerflow flag Set.
CLC CLear Carry flag.
CLD CLear Decimal mode.
CLI CLear Interrupt disable flag.
CLV CLear oVerflow flag.
CMP CoMPare memory and accumulator.
CPX ComPare memory and index X.
CPY ComPare memory and index Y.
DEC DECrement memory by one.
DEX DEcrement register S by one.
DEY DEcrement register Y by one.
EOR Exclusive OR memory with accumulator.
INC INCrement memory by one.
INX INcrement register X by one.
INY INcrement register Y by one.
JMP JuMP to new memory location.
JSR Jump to SubRoutine.
LDA LoaD Accumulator from memory.
LDX LoaD X register from memory.
LDY LoaD Y register from memory.
LSR Logical Shift Right one bit.
NOP No OPeration.
ORA OR memory with Accumulator.
PHA PusH Accumulator on stack.
PHP PusH Processor status on stack.
PLA PuLl Accumulator from stack.
PLP PuLl Processor status from stack.
ROL ROtate Left one bit.
ROR ROtate Right one bit.
RTI ReTurn from Interrupt.
RTS ReTurn from Subroutine.
SBC SuBtract with Carry.
SEC SEt Carry flag.
SED SEt Decimal mode.
SEI SEt Interrupt disable flag.
STA STore Accumulator in memory.
STX STore Register X in memory.
STY STore Register Y in memory.
TAX Transfer Accumulator to register X.
TAY Transfer Accumulator to register Y.
TSX Transfer Stack pointer to register X.
TXA Transfer register X to Accumulator.
TXS Transfer register X to Stack pointer.
TYA Transfer register Y to Accumulator.

The following is a small program which has been written using the 6500 family's instruction set. The purpose of the program is to calculate the sum of the 10 numbers which have been placed into memory started at address 0200 hexadecimal.

Here are the 10 numbers in memory ( which are printed in blue ) along with the memory location that the sum will be stored into ( which is printed in red ). 0200 here is the address in memory of the first number.

```
0200  01 02 03 04 05 06 07 08 - 09 0A 00 00 00 00 00 00  ................

```

Here is a program that will calculate the sum of these 10 numbers:

```
0250  A2 00     LDX #00h
0252  A9 00     LDA #00h
0254  18        CLC
0255  7D 00 02  ADC 0200h,X
0258  E8        INX
0259  E0 0A     CPX #0Ah
025B  D0 F8     BNE 0255h
025D  8D 0A 02  STA 020Ah
0260  00        BRK
...
```

After the program was executed, the sum it calculated was stored in memory. The sum was determined to be 37 hex ( which is 55 decimal ) and it is shown here printed in red:

```
0200  01 02 03 04 05 06 07 08 - 09 0A 37 00 00 00 00 00  ..........7.....
```

Of course, you are not expected to understand how this assembly language program works. The purpose for showing it to you is so you can see what a program that uses a microprocessor's instruction set looks like.

## Low Level Languages And High Level Languages

Even though programmers are able to program a computer using the instructions in its instruction set, this is a tedious task. The early computer programmers wanted to develop programs in a language that was more like a natural language, English for example, than the machine language that microprocessors understand. Machine language is considered to be a low level languages because it was designed to be simple so that it could be easily executed by the circuits in a microprocessor.

Programmers then figured out ways to use low level languages to create the high level languages that they wanted to program in. This is when languages like FORTRAN ( in 1957 ), ALGOL ( in 1958 ), LISP ( in 1959 ), COBOL ( in 1960 ), BASIC ( in 1964 ) and C ( 1972 ) were created. Ultimately, a microprocessor is only capable of understanding machine language and therefore all programs that are written in a high level language must be converted into machine language before they can be executed by a microprocessor.

The rules that indicate how to properly type in code for a given programming language are called syntax rules. If a programmer does not follow the language's syntax rules when typing in a program, the software that transforms the source code into machine language will become confused and then issue what is called a syntax error.

As an example of what a syntax error might look like, consider the word 'print'. If the word 'print' was a command in a given program language, and the programmer typed 'pvint' instead of 'print', this would be a syntax error.

## 18.6 Compilers And Interpreters

There are two types of programs that are commonly used to convert a higher level language into machine language. The first kind of program is called a compiler and it takes a high-level language's source code ( which is usually in typed form ) as its input and converts it into machine language. After the machine language equivalent of the source code has been generated, it can be loaded into a computer's memory and executed. The compiled version of a program can also be saved on a storage device and loaded into a computer's memory whenever it is needed.

The second type of program that is commonly used to convert a high-level language into machine language is called an interpreter. Instead of converting source code into machine language like a compiler does, an interpreter reads the source code ( usually one line at a time ), determines what actions this line of source code is suppose to accomplish, and then it performs these actions. It then looks at the next line of source code underneath the one it just finished interpreting, it determines what actions this next line of code wants done, it performs these actions, and so on.

Thousands of computer languages have been created since the 1940's, but there are currently around 2 to 3 hundred historically important languages. Here is a link to a website that lists a number of the historically important computer languages: http://en.wikipedia.org/wiki/Timeline_of_programming_languages

## 18.7 Algorithms

A computer programmer certainly needs to know at least one programming language, but when a programmer solves a problem, they do it at a level that is higher in abstraction than even the more abstract computer languages.

After the problem is solved, then the solution is encoded into a programming language. It is almost as if a programmer is actually two people. The first person is the problem solver and the second person is the coder.

For simpler problems, many programmers create algorithms in their minds and encode these algorithm directly into a programming language. They switch back and forth between being the problem solver and the coder during this process.

With more complex programs, however, the problem solving phase and the coding phase are more distinct. The algorithm which solves a given problem is is developed using means other than a programming language and then it is recored in a document. This document is then passed from the problem solver to the coder for encoding into a programming language.

The first thing that a problem solver will do with a problem is to analyze it. This is an extremely important step because if a problem is not analyzed, then it can not be properly solved. To analyze something means to break it down into its component parts and then these parts are studied to determine how they work. A well known saying is 'divide and conquer' and when a difficult problem is analyzed, it is broken down into smaller problems which are each simpler to solve than the overall problem. The problem solver then develops an algorithm to solve each of the simpler problems and, when these algorithms are combined, they form the solution to the overall problem.

An algorithm ( pronounced al-gor-rhythm ) is a sequence of instructions which describe how to accomplish a given task. These instructions can be expressed in various ways including writing them in natural languages ( like English ), drawing diagrams of them, and encoding them in a programming language.

The concept of an algorithm came from the various procedures that mathematicians developed for solving mathematical problems, like calculating the sum of 2 numbers or calculating their product.

Algorithms can also be used to solve more general problems. For example, the following algorithm could have been followed by a person who wanted to solve the garage sizing problem using paper models:

1) Measure the length and width of each item that will be placed into the garage using metric units and record these measurements.

2) Divide the measurements from step 1 by 100 then cut out pieces of paper that match these dimensions to serve as models of the original items.

3) Cut out a piece of paper which is 1.5 times as long as the model of the largest car and 3 times wider than it to serve as a model of the garage floor.

4) Locate where the garage doors will be placed on the model of the garage floor,

mark the locations with a pencil, and place the models of both cars on top of the model of the garage floor, just within the perimeter of the paper and between the two pencil marks.

5) Place the models of the items on top of the model of the garage floor in the empty space that is not being occupied by the models of the cars.

6) Move the models of the items into various positions within this empty space to determine how well all the items will fit within this size garage.

7) If the fit is acceptable, go to step 10.

8) If there is not enough room in the garage, increase the length dimension, the width dimension ( or both dimensions ) of the garage floor model by 10%, create a new garage floor model, and go to step 4.

9) If there is too much room in the garage, decrease the length dimension, the width dimension ( or both dimensions ) of the garage model by 10%, create a new garage floor model, and go to step 4.

10) Measure the length and width dimensions of the garage floor model, multiply these dimensions by 100, and then build the garage using these larger dimensions.

As can be seen with this example, an algorithm often contains a significant number of steps because it needs to be detailed enough so that it leads to the desired solution. After the steps have been developed and recorded in a document, however, they can be followed over and over again by people who need to solve the given problem.

## *18.8 Computation*

It is fairly easy to understand how a human is able to follow the steps of an algorithm, but it is more difficult to understand how computer can perform these steps when its microprocessor is only capable of executing simple machine language instructions.

In order to understand how a microprocessor is able to perform the steps in an algorithm, one must first understand what computation ( which is also known as calculation ) is. Lets search for some good definitions of each of these words on the Internet and read what they have to say.”

Here are two definitions for the word computation:

1) The manipulation of numbers or symbols according to fixed rules. Usually applied to the operations of an automatic electronic computer, but by extension

to some processes performed by minds or brains. ( www.informatics.susx.ac.uk/books/computers-and-thought/gloss/node1.html )

2) A computation can be seen as a purely physical phenomenon occurring inside a closed physical system called a computer. Examples of such physical systems include digital computers, quantum computers, DNA computers, molecular computers, analog computers or wetware computers. ( www.informatics.susx.ac.uk/books/computers-and-thought/gloss/node1.html )

These two definitions indicate that computation is the "manipulation of numbers or symbols according to fixed rules" and that it "can be seen as a purely physical phenomenon occurring inside a closed physical system called a computer." Both definitions indicate that the machines we normally think of as computers are just one type of computer and that other types of closed physical systems can also act as computers. These other types of computers include DNA computers, molecular computers, analog computers, and wetware computers ( or brains ).

The following two definitions for calculation shed light on the kind of rules that normal computers, brains, and other types of computers use:

1) A calculation is a deliberate process for transforming one or more inputs into one or more results. ( en.wikipedia.org/wiki/Calculation )

2) Calculation: the procedure of calculating; determining something by mathematical or logical methods ( wordnet.princeton.edu/perl/webwn )

These definitions for calculation indicate that it "is a deliberate process for transforming one or more inputs into one or more results" and that this is done "by mathematical or logical methods". We do not yet completely understand what mathematical and logical methods brains use to perform calculations, but rapid progress is being made in this area.

The second definition for calculation uses the word logic and this word needs to be defined before we can proceed:

The logic of a system is the whole structure of rules that must be used for any reasoning within that system. Most of mathematics is based upon a well-understood structure of rules and is considered to be highly logical. It is always necessary to state, or otherwise have it understood, what rules are being used before any logic can be applied. ( ddi.cs.uni-potsdam.de/Lehre/TuringLectures/MathNotions.htm )

Reasoning is the process of using predefined rules to move from one point in a system to another point in the system. For example, when a person adds 2 numbers together on a piece of paper, they must follow the rules of the addition algorithm in order to obtain a correct sum. The addition algorithm's rules are its logic and, when someone applies these rules during a calculation, they are reasoning with the rules.

Lets now apply these concepts to the question about how a computer can perform the steps of an algorithm when its microprocessor is only capable of executing simple machine language instructions. When a person develops an algorithm, the steps in the algorithm are usually stated as high-level tasks which do not contain all of the smaller steps that are necessary to perform each task.

For example, a person might write a step that states "Drive from New York to San Francisco." This large step can be broken down into smaller steps that contain instructions such as "turn left at the intersection, go west for 10 kilometers, etc." If all of the smaller steps in a larger step are completed, then the larger step is completed too.

A human that needs to perform this large driving step would usually be able to figure out what smaller steps need to be performed in order accomplish it. Computers are extremely stupid, however, and before any algorithm can be executed on a computer, the algorithm's steps must be broken down into smaller steps, and these smaller steps must be broken down into even small steps, until the steps are simple enough to be performed by the instruction set of a microprocessor.

Sometimes only a few smaller steps are needed to implement a larger step, but sometimes hundreds or even thousands of smaller steps are required. Hundreds or thousands of smaller steps will translate into hundreds or thousands of machine language instructions when the algorithm is converted into machine language.

If machine language was the only language that computers could be programmed in, then most algorithms would be too large to be placed into a computer by a human. An algorithm that is encoded into a high-level language, however, does not need to be broken down into as many smaller steps as would be needed with machine language. The hard work of further breaking down an algorithm that has been encoded into a high-level language is automatically done by either a compiler or an interpreter. This is why most of the time, programmers use a high-level language to develop in instead of machine language.

## 12.11 Diagrams Can Be Used To Record Algorithms

Earlier it was mentioned that not only can an algorithm can be recorded in a natural language like English but it can also be recorded using diagrams. You may be surprised to learn, however, that a whole diagram-based language has been created which allows all aspects of a program to be designed by 'problem solvers', including the algorithms that a program uses. This language is call UML which stands for Unified Modeling Language. One of UML's diagrams is called an Activity diagram and it can be used to show the sequence of steps (or activities) that are part of some piece of logic. The following is an example which shows how an algorithm can be represented in an Activity diagram.

12.12 Calculating The Sum Of The Numbers Between 1 And 10

The first thing that needs to be done with a problem before it can be analyzed and solved is to describe it clearly and accurately. Here is a short description for the problem we will solve with an algorithm:

Description: In this problem, the sum of the numbers between 1 and 10 inclusive needs to be determined.

Inclusive here means that the numbers 1 and 10 will be included in the sum. Since this is a fairly simple problem we will not need to spend too much time analyzing it. Drawing 11 shows an algorithm for solving this problem that has been placed into an Activity diagram.

An algorithms and its Activity diagram are developed at the same time. During the development process, variables are created as needed and their names are usually recorded in a list along with their descriptions. The developer periodically starts at the entry point and walks through the logic to make sure it is correct. Simulation boxes are placed next to each variable so that they can be use to record and update how the logic is changing the variable's values. During a walk-through, errors are usually found and these need to be fixed by moving flow arrows and adjusting the text that is inside of the activity rectangles.

When the point where no more errors in the logic can be found, the developer can stop being the problem solver and pass the algorithm over to the coder so it can be encoded into a programming language.

## *18.9 The Mathematics Part Of Mathematics Computing Systems*

Mathematics has been described as the "science of patterns" 2. Here is a definition for pattern:

1) Systematic arrangement...
(http://www.answers.com/topic/pattern)

And here is a definition for system:

1) A group of interacting, interrelated, or interdependent elements forming a complex whole.

2) An organized set of interrelated ideas or principles.
(http://www.answers.com/topic/system)

Therefore, mathematics can be though of as a science that deals with the systematic properties of physical and nonphysical objects. The reason that mathematics is so powerful is that all physical and nonphysical objects posses systematic properties and therefore, mathematics is a means by which these objects can be understood and manipulated.

The more mathematics a person knows, the more control they are able to have over the physical world. This makes mathematics one of the most useful and exciting areas of knowledge a person can possess.

Traditionally, learning mathematics also required learning the numerous tedious and complex algorithms that were needed to perform written calculations with mathematics. Usually over 50% of the content of the typical traditional math textbook is devoted to teaching writing-based algorithms and an even higher percentage of the time a person spends working through a textbook is spent manually working these algorithms.

For most people, learning and performing tedious, complex written-calculation algorithms is so difficult and mind-numbingly boring that they never get a chance to see that the "mathematics" part of mathematics is extremely exciting, powerful, and beautiful.

The bad news is that writing-based calculation algorithms will always be tedious, complex, and boring. The good news is that the invention of mathematics computing environments has significantly reduced the need for people to use writing-based calculation algorithms.

Notes:

+ Create link to "computation".
+ Create link to "algorithm".
+

MathPiper information.

----
MathPiper can evaluate limits (which are the beginnings of calculus). The syntax is:

Limit(var, val) expr

...Where "var" is the variable that approaches some value, "val" is the value it approaches, and "expr" is the expression whose limit you want to find as var approaches val. Let's use the following ultra-simple limit calculation as an example:

```
Limit(x,2) x
```

This line says "find the limit of x as x approaches 2". The answer, obviously, is 2. The next one is a little trickier:

```
Limit(x,1) 5*(x-1)/(x-1)
```

Producing a direct result for the expression is impossible, because it creates a divide-by-zero situation. (Note that a lot of calculus limits are used explicitly because they're intended to evaluate expressions that involve dividing by zero.) However, if you consider the expression (x-1) on its own, you'll realize that we are multiplying 5 by this value, then immediately dividing the result by this same value. Since multiplying something by any value and then immediately dividing by the same value should, in general, leave the original number unchanged, we see that even as x approaches very close to 1, the expression remains 5; the expression doesn't become undefined until x is exactly 1. Hence, the limit is 5.

Limits are cool in this way, because they allow you to evaluate things involving division by zero, but they have their limits (pun not intended). The following MathPiper line will still yield "Undefined":

```
Limit(x,1) x/0
```

Moving on from limits, you can do calculus derivatives with MathPiper using the D function, like this:

```
D(x) x*2
D(x) x^2
```

Doing indefinite integrals is pretty straightforward:

```
Integrate(x) x*2
Integrate(x) x^2
Integrate(x) x
```

You can add the left- and right-hand sides of a range to calculate a definite integral, as well:

```
Integrate (x, 1, 2) x
Integrate (x, 2, 3) x
Integrate (x, 1, 2) x*2
Integrate (x, 2, 3) x*2
```

----

2^Infinity

Oddly enough, however, MathPiper does *NOT* contain e (the base of the natural logarithm) as a constant. However, you can use e by making use of the Exp() function. This function calculates e raised to the power of its argument; for example, the following calculates e^2:

Exp(2)

Based on this, you can use Exp(1) to represent e. Or, better yet, you can simply use the following line to define your own e, and then just use "e" in the future:

Set(e,Exp(1))

----

Thus, "This text" is what is called one token, surrounded by quotes, in MathPiper.

----

The usual notation people use when writing down a calculation is called the infix notation, and you can readily recognize it, as for example 2+3 and 3*4. Prefix operators also exist. These operators come before an expression, like for example the unary minus sign (called unary because it accepts one argument), -(3*4). In addition to prefix operators there are also postfix operators, like the exclamation mark to calculate the factorial of a number, 10!.

----

Functions usually have the form f(), f(x) or f(x,y,z,...) depending on how many arguments the function accepts. Functions always return a result.

----

Evaluating functions can be thought of as simplifying an expression as much as possible. Sometimes further simplification is not possible and a function returns itself unsimplified, like taking the square root of an integer Sqrt(2). A reduction to a number would be an approximation. We explain elsewhere how to get MathPiper to simplify an expression to a number.

----

MathPiper allows for use of the infix notation, but with some additions. Functions can be "bodied", meaning that the last argument is written past the close bracket. An example is ForEach, where we write ForEach(item, 1 .. 10) Echo(item);. Echo(item) is the last argument to the function ForEach.

----

{a,b,c}[2] should return b, as b is the second element in the list (MathPiper starts counting from 1 when accessing elements). The same can be done with strings: "abc"[2]

----

And finally, function calls can be grouped together, where they get executed one at a time, and the result of executing the last expression is returned. This is done through square brackets, as [ Echo("Hello"); Echo("World"); True; ];, which first writes Hello to screen, then World on the next line, and then returns True.

----

A session can be restarted (forgetting all previous definitions and results) by typing restart. All memory is erased in that case.

----

Statements should end with a semicolon ; although this is not required in interactive sessions (MathPiper will append a semicolon at end of line to finish the statement).

----

Commands spanning multiple lines can (and actually have to) be entered by using a trailing backslash \ at end of each continued line. For example, clicking on 2+3+ will result in an error, but entering the same with a backslash at the end and then entering another expression will concatenate the two lines and evaluate the concatenated input.

----

Incidentally, any text MathPiper prints without a prompt is either a message printed by a function as a side-effect, or an error message. Resulting values of expressions are always printed after an Result> prompt.

----

A numeric vs. a symbolic calculator.

----

MathPiper as a symbolic calculator

We are ready to try some calculations. MathPiper uses a C-like infix syntax and is case-sensitive. Here are some exact manipulations with fractions for a start:
1/14+5/21*(30-(1+1/2)*5^2);

The standard scripts already contain a simple math library for symbolic simplification of basic algebraic functions. Any names such as x are treated as independent, symbolic variables and are not evaluated by default. Some examples to try:

* 0+x
* x+1*y
* Sin(ArcSin(alpha))+Tan(ArcTan(beta))

Note that the answers are not just simple numbers here, but actual expressions. This is where MathPiper shines. It was built specifically to do calculations that have expressions as answers.

----

In MathPiper after a calculation is done, you can refer to the previous result with %. For example, we could first type (x+1)*(x-1), and then decide we would like to see a simpler version of that expression, and thus type Simplify(%), which should result in x^2-1.

The special operator % automatically recalls the result from the previous line.

----

The function Simplify attempts to reduce an expression to a simpler form.
----
Note that standard function names in MathPiper are typically capitalized. Multiple capitalization such as ArcSin is sometimes used.
----
The underscore character _ is a reserved operator symbol and cannot be part of variable or function names.
----
MathPiper offers some more powerful symbolic manipulation operations. A few will be shown here to wetten the appetite.

Some simple equation solving algorithms are in place:

* Solve(x/(1+x) == a, x);
* Solve(x^2+x == 0, x);
* Solve(a+x*y==z,x);

(Note the use of the == operator, which does not evaluate to anything, to denote an "equation" object.)
----
Symbolic manipulation is the main application of MathPiper.
----
This is a small tour of the capabilities MathPiper currently offers. Note that this list of examples is far from complete. MathPiper contains a few hundred commands, of which only a few are shown here.

* Expand((1+x)^5); (expand the expression into a polynomial)
* Limit(x,0) Sin(x)/x; (calculate the limit of Sin(x)/x as x approaches zero)
* Newton(Sin(x),x,3,0.0001); (use Newton's method to find the value of x near 3 where Sin(x) equals zero, numerically, and stop if the result is closer than 0.0001 to the real result)
* DiagonalMatrix({a,b,c}); (create a matrix with the elements specified in the vector on the diagonal)
* Integrate(x,a,b) x*Sin(x); (integrate a function over variable x, from a to b)
* Factor(x^2-1); (factorize a polynomial)
* Apart(1/(x^2-1),x); (create a partial fraction expansion of a polynomial)
* Simplify((x^2-1)/(x-1)); (simplification of expressions)
* CanProve( (a And b) Or (a And Not b) ); (special-purpose simplifier that tries to simplify boolean expressions as much as possible)
* TrigSimpCombine(Cos(a)*Sin(b)); (special-purpose simplifier that tries to transform trigonometric expressions into a form where there are only additions of trigonometric functions involved and no multiplications)
----
MathPiper can deal with arbitrary precision numbers. It can work with large integers, like 20! (The ! means factorial, thus 1*2*3*...*20).
----

As we saw before, rational numbers will stay rational as long as the numerator and denominator are integers, so 55/10 will evaluate to 11/2. You can override this behavior by using the numerical evaluation function N(). For example, N(55/10) will evaluate to 5.5 . This behavior holds for most math functions. MathPiper will try to maintain an exact answer (in terms of integers or fractions) instead of using floating point numbers, unless N() is used. Where the value for the constant pi is needed, use the built-in variable Pi. It will be replaced by the (approximate) numerical value when N(Pi) is called.

----

MathPiper knows some simplification rules using Pi (especially with trigonometric functions).

----

Thus N(1/234) returns a number with the current default precision (which starts at 20 digits)

----

Note that we need to enter N() to force the approximate calculation, otherwise the fraction would have been left unevaluated.

----

Taking a derivative of a function was amongst the very first of symbolic calculations to be performed by a computer, as the operation lends itself surprisingly well to being performed automatically.

----

D is a bodied function, meaning that its last argument is past the closing brackets. Where normal functions are called with syntax similar to f(x,y,z), a bodied function would be called with a syntax f(x,y)z. Here are two examples of taking a derivative:

* D(x) Sin(x); (taking a derivative)
* D(x) D(x) Sin(x); (taking a derivative twice)

----

Analytic functions

Many of the usual analytic functions have been defined in the MathPiper library. Examples are Exp(1), Sin(2), ArcSin(1/2), Sqrt(2). These will not evaluate to a numeric result in general, unless the result is an integer, like Sqrt(4). If asked to reduce the result to a numeric approximation with the function N, then MathPiper will do so, as for example in N(Sqrt(2),50).

----

Variables

MathPiper supports variables. You can set the value of a variable with the := infix operator, as in a:=1;. The variable can then be used in expressions, and everywhere where it is referred to, it will be replaced by its value, a.

----

To clear a variable binding, execute Clear(a);. A variable will evaluate to itself after a call to clear it (so after the call to clear a above, calling a should now return a). This is one of the properties of the evaluation scheme of MathPiper; when some object can not be evaluated or transformed any further, it is returned as the final result.

----

## Functions

The := operator can also be used to define simple functions: f(x):=2*x*x. will define a new function, f, that accepts one argument and returns twice the square of that argument. This function can now be called, f(a) (Note:tk: called means executing the function). You can change the definition of a function by defining it again.

----

One and the same function name such as f may define different functions if they take different numbers of arguments. One can define a function f which takes one argument, as for example f(x):=x^2;, or two arguments, f(x,y):=x*y;. If you clicked on both links, both functions should now be defined, and f(a) calls the one function whereas f(a,b) calls the other.

----

MathPiper is very flexible when it comes to types of mathematical objects. (Note: exactly which types are being referred to? ). Functions can in general accept or return any type of argument.

----

## Boolean expressions and predicates

MathPiper predefines True and False as boolean values. Functions returning boolean values are called predicates. For example, IsNumber() and IsInteger() are predicates defined in the MathPiper environment. For example, try IsNumber(2+x);, or IsInteger(15/5);.

----

There are also comparison operators. Typing 2 > 1 would return True.

----

You can also use the infix operators And and Or, and the prefix operator Not, to make more complex boolean expressions. For example, try True And False, True Or False, True And Not(False).

----

## Strings and lists

In addition to numbers and variables, MathPiper supports strings and lists. Strings are simply sequences of characters enclosed by double quotes, for example: "this is a string with \"quotes\" in it".

----

Lists are ordered groups of items, as usual. MathPiper represents lists by putting the objects between braces and separating them with commas. The list consisting of objects a, b, and c could be entered by typing {a,b,c}.

----

In MathPiper, vectors are represented as lists and matrices as lists of lists.

----
Items in a list can be accessed through the [ ] operator. The first element has index one. Examples: when you enter uu:={a,b,c,d,e,f}; then uu[2]; evaluates to b, and uu[2 .. 4]; evaluates to {b,c,d}.
----
The "range" expression 2 .. 4 evaluates to {2,3,4}. Note that spaces around the .. operator are necessary, or else the parser will not be able to distinguish it from a part of a number.
----
Lists evaluate their arguments, and return a list with results of evaluating each element. So, typing {1+2,3}; would evaluate to {3,3}.
----
The idea of using lists to represent expressions dates back to the language LISP developed in the 1970's. From a small set of operations on lists, very powerful symbolic manipulation algorithms can be built.
----
Lists can also be used as function arguments when a variable number of arguments are necessary.
----
Let's try some list operations now. First click on m:={a,b,c}; to set up an initial list to work on. Then click on links below:

* Length(m); (return the length of a list)
* Reverse(m); (return the string reversed)
* Concat(m,m); (concatenate two strings)
* m[1]:=d; (setting the first element of the list to a new value, d, as can be verified by evaluating m)

----
Writing simplification rules

Mathematical calculations require versatile transformations on symbolic quantities. Instead of trying to define all possible transformations, MathPiper provides a simple and easy to use pattern matching scheme for manipulating expressions according to user-defined rules.
----
MathPiper itself is designed as a small core engine executing a large library of rules to match and replace patterns.
----
One simple application of pattern-matching rules is to define new functions. (This is actually the only way MathPiper can learn about new functions.) Note:tk:what does this mean?
----
----
As an example, let's define a function f that will evaluate factorials of non-negative integers. We will define a predicate to check whether our argument is indeed a non-negative integer, and we will use this predicate and the obvious

recursion f(n)=n*f(n-1) if n>0 and 1 if n=0 to evaluate the factorial.
----
We start with the simple termination condition, which is that f(n) should return one if n is zero:

* 10 # f(0) <-- 1;

You can verify that this already works for input value zero, with f(0).
----
Now we come to the more complex line,

* 20 # f(n_IsIntegerGreaterThanZero) <-- n*f(n-1);
----
Now we realize we need a function IsGreaterThanZero, so we define this function, with

* IsIntegerGreaterThanZero(_n) <-- (IsInteger(n) And n>0);

You can verify that it works by trying f(5), which should return the same value as 5!.
----
In the above example we have first defined two "simplification rules" for a new function f().
----
Then we realized that we need to define a predicate IsIntegerGreaterThanZero(). A predicate equivalent to IsIntegerGreaterThanZero() is actually already defined in the standard library and it's called IsPositiveInteger, so it was not necessary, strictly speaking, to define our own predicate to do the same thing. We did it here just for illustration purposes.
----
The first two lines recursively define a factorial function f(n)=n*(n-1)*...*1. The rules are given precedence values 10 and 20, so the first rule will be applied first.
----
Incidentally, the factorial is also defined in the standard library as a postfix operator ! and it is bound to an internal routine much faster than the recursion in our example.
----
The example does show how to create your own routine with a few lines of code. One of the design goals of MathPiper was to allow precisely that, definition of a new function with very little effort.
----
The operator <-- defines a rule to be applied to a specific function. (The <-- operation cannot be applied to an atom.)
----
The _n in the rule for IsIntegerGreaterThanZero() specifies that any object which

happens to be the argument of that predicate is matched and assigned to the local variable n. The expression to the right of <-- can use n (without the underscore) as a variable.

----

Now we consider the rules for the function f. The first rule just specifies that f(0) should be replaced by 1 in any expression.

----

The second rule is a little more involved. n_IsIntegerGreaterThanZero is a match for the argument of f, with the proviso that the predicate IsIntegerGreaterThanZero(n) should return True, otherwise the pattern is not matched.

----

The underscore operator is to be used only on the left hand side of the rule definition operator <--.

----

Note:tk:this needs to be studied further.

There is another, slightly longer but equivalent way of writing the second rule:

```
* 20 # f(_n)_(IsIntegerGreaterThanZero(n)) <-- n*f(n-1);
```

The underscore after the function object denotes a "postpredicate" that should return True or else there is no match. This predicate may be a complicated expression involving several logical operations, **unlike the simple checking of just one predicate in the n_IsIntegerGreaterThanZero construct**. The postpredicate can also use the variable n (without the underscore).

----

Precedence values for rules are given by a number followed by the # infix operator (and the transformation rule after it). This number determines the ordering of precedence for the pattern matching rules, with 0 the lowest allowed precedence value, i.e. rules with precedence 0 will be tried first.

----

Multiple rules can have the same number: this just means that it doesn't matter what order these patterns are tried in.

----

If no number is supplied, 0 is assumed.

----

In our example, the rule f(0) <-- 1 must be applied earlier than the recursive rule, or else the recursion will never terminate.

----

But as long as there are no other rules concerning the function f, the assignment of numbers 10 and 20 is arbitrary, and they could have been 500 and 501 just as well.

----

It is usually a good idea however to keep some space between these numbers, so you have room to insert new transformation rules later on.

----

Predicates can be combined: for example, {IsIntegerGreaterThanZero()} could also have been defined as:

* 10 # IsIntegerGreaterThanZero(n_IsInteger)_(n>0) <-- True;
* 20 # IsIntegerGreaterThanZero(_n) <-- False;

The first rule specifies that if n is an integer, and is greater than zero, the result is True, and the second rule states that otherwise (when the rule with precedence 10 did not apply) the predicate returns False.

----

In the above example, the expression n > 0 is added after the pattern and allows the pattern to match only if this predicate return True. This is a useful syntax for defining rules with complicated predicates. There is no difference between the rules F(n_IsPositiveInteger) <--... and F(_n)_(IsPositiveInteger(n)) <-- ... except that the first syntax is a little more concise.

----

The left hand side of a rule expression has the following form:
precedence # pattern _ postpredicate <-- replacement ;
The optional precedence must be a positive integer.

----

Some more examples of rules (not made clickable because their equivalents are already in the basic MathPiper library):

* 10 # _x + 0 <-- x;
* 20 # _x - _x <-- 0;
* ArcSin(Sin(_x)) <-- x;

**The last rule has no explicit precedence specified in it (the precedence zero will be assigned automatically by the system)**.

----

----

MathPiper will first try to match the pattern as a template.

----

Names preceded or followed by an underscore can match any one object: a number, a function, a list, etc.

----

MathPiper will assign the relevant variables as local variables within the rule, and try the predicates as stated in the pattern.

----

The post-predicate (defined after the pattern) is tried after all these matched.

----

As an example, the simplification rule _x - _x <--0 specifies that the two objects at left and at right of the minus sign should be the same for this transformation rule to apply.

----

Local simplification rules

Sometimes you have an expression, and you want to use specific simplification rules on it that should not be universally applied. This can be done with the /: and the /:: operators.
----
Suppose we have the expression containing things such as Ln(a*b), and we want to change these into Ln(a)+Ln(b). The easiest way to do this is using the /: operator as follows:

* Sin(x)*Ln(a*b) (example expression without simplification)
* Sin(x)*Ln(a*b) /: { Ln(_x*_y) <- Ln(x)+Ln(y) } (with instruction to simplify the expression)

----
A whole list of simplification rules can be built up in the list, and they will be applied to the expression on the left hand side of /:.
----
Note that for these local rules, <- should be used instead of <--. Using latter would result in a global definition of a new transformation rule on evaluation, which is not the intention.
----
The /: operator traverses an expression from the top down, trying to apply the rules from the beginning of the list of rules to the end of the list of rules. If no rules can be applied to the whole expression, it will try the sub-expressions of the expression being analyzed.
----
It might be sometimes necessary to use the /:: operator, which repeatedly applies the /: operator until the result does not change any more. Caution is required, since rules can contradict each other, and that could result in an infinite loop. To detect this situation, just use /: repeatedly on the expression. The repetitive nature should become apparent.
----
Looping can be done with the function ForEach. There are more options, but ForEach is the simplest to use for now and will suffice for this turorial. The statement form ForEach(x, list) body executes its body for each element of the list and assigns the variable x to that element each time.
----
The statement form While(predicate) body repeats execution of the expression represented by body until evaluation of the expression represented by predicate returns False.
----
This example loops over the integers from one to three, and writes out a line for each, multiplying the integer by 3 and displaying the result with the function Echo: ForEach(x,1 .. 5) Echo(x," times 3 equals ",3*x);
----
Compound statements

Multiple statements can be grouped together using the [ and ] brackets. The compound [a; Echo("In the middle"); 1+2;]; evaluates a, then the echo command, and finally evaluates 1+2, **and returns the result of evaluating the last statement 1+2**.

----

A variable can be declared local to a compound statement block by the function Local(var1, var2,...). For example, if you execute [Local(v);v:=1+2;v;]; the result will be 3. The program body created a variable called v, assigned the value of evaluating 1+2 to it, and made sure the contents of the variable v were returned. If you now evaluate v afterwards you will notice that the variable v is not bound to a value any more. The variable v was defined locally in the program body between the two square brackets [ and ].

----

Conditional execution is implemented by the If(predicate, body1, body2) function call. If the expression predicate evaluates to True, the expression represented by body1 is evaluated, otherwise body2 is evaluated, and the corresponding value is returned. For example, the absolute value of a number can be computed with: f(x) := If(x < 0,-x,x); (note that there already is a standard library function that calculates the absolute value of a number).

----

Variables can also be made to be local to a small set of functions, with LocalSymbols(variables) body.

----

For example, the following code snippet: LocalSymbols(a,b) [a:=0;b:=0; inc():=[a:=a+1;b:=b-1;show();]; show():=Echo("a = ",a," b = ",b); ]; defines two functions, inc and show. Calling inc() repeatedly increments a and decrements b, and calling show() then shows the result (the function "inc" also calls the function "show", but the purpose of this example is to show how two functions can share the same variable while the outside world cannot get at that variable). The variables are local to these two functions, as you can see by evaluating a and b outside the scope of these two functions.

----

This feature is very important when writing a larger body of code, where you want to be able to guarantee that there are no unintended side-effects due to two bits of code defined in different files accidentally using the same global variable.

----

To illustrate these features, let us create a list of all even integers from 2 to 20 and compute the product of all those integers except those divisible by 3. (What follows is not necessarily the most economical way to do it in MathPiper.)

```

[
  Local(L,i,answer);
  L:={};
```

```
 i:=2;
 /* Make a list of all even integers from 2 to 20 */
 While (i<=20)
 [
   L := Append(L,i);
   i := i + 2;
 ];
 /* Now calculate the product of all of
    these numbers that are not divisible by 3 */
 answer := 1;
 ForEach(i,L)
   If (Mod(i, 3)!=0, answer := answer * i);
 /* And return the answer */
 answer;
];
```

----

We used a shorter form of If(predicate, body) with only one body which is executed when the condition holds. If the condition does not hold, this function call returns False.

----

We also introduced comments, which can be placed between /* and */. MathPiper will ignore anything between those two.

----

When putting a program in a file you can also use //. Everything after // up until the end of the line will be a comment.

----

Also shown is the use of the While function. Its form is While (predicate) body. While the expression represented by predicate evaluates to True, the expression represented by body will keep on being evaluated.

----

The above example is not the shortest possible way to write out the algorithm. It is written out in a procedural way, where the program explains step by step what the computer should do. There is nothing fundamentally wrong with the approach of writing down a program in a procedural way, but the symbolic nature of MathPiper also allows you to write it in a more concise, elegant, compact way, by combining function calls.

----

There is nothing wrong with procedural style, but there is a more 'functional' approach to the same problem would go as follows below.

----

The advantage of the functional approach is that it is shorter and more concise (the difference is cosmetic mostly).

----

Before we show how to do the same calculation in a functional style, we need to explain what a "pure function" is, as you will need it a lot when programming in a functional style.

----

We will jump in with an example that should be self-explanatory. Consider the expression Lambda({x,y},x+y). This has two arguments, the first listing x and y, and the second an expression. We can use this construct with the function Apply as follows:

----

Apply(Lambda({x,y},x+y),{2,3}). The result should be 5, the result of adding 2 and 3.

----

The expression starting with Lambda is essentially a prescription for a specific operation, where it is stated that it accepts 2 arguments, and returns the two arguments added together.

----

In this case, since the operation was so simple, we could also have used the name of a function to apply the arguments to, the addition operator in this case Apply("+",{2,3}).

----

When the operations become more complex however, the Lambda construct becomes more useful.

----

Now we are ready to do the same example using a functional approach. First, let us construct a list with all even numbers from 2 to 20. For this we use the .. operator to set up all numbers from one to ten, and then multiply that with two: 2*(1 .. 10).

----

Now we want an expression that returns all the even numbers up to 20 which are not divisible by 3.

----

For this we can use Select, which takes as first argument a predicate that should return True if the list item is to be accepted, and false otherwise, and as second argument the list in question: Select(Lambda({n},Mod(n,3)!=0),2*(1 .. 10)). The numbers 6, 12 and 18 have been correctly filtered out.

----

Here you see one example of a pure function where the operation is a little bit more complex.

----

All that remains is to factor the items in this list. For this we can use UnFlatten.

----

Two examples of the use of UnFlatten are UnFlatten({a,b,c},"*",1) and UnFlatten({a,b,c},"+",0). The 0 and 1 are a base element to start with when grouping the arguments in to an expression (hence it is zero for addition and 1 for multiplication).

----

Now we have all the ingredients to finally do the same calculation we did above in a procedural way, but this time we can do it in a functional style, and thus captured in one concise single line:

UnFlatten(Select(Lambda({n},Mod(n,3)!=0),2*(1 .. 10)),"*",1).

As was mentioned before, the choice between the two is mostly a matter of style.

----

## Macros

One of the powerful constructs in MathPiper is the construct of a macro. In its essence, a macro is a prescription to create another program before executing the program.

----

An example perhaps explains it best. Evaluate the following expression Macro(for,{st,pr,in,bd}) [(@st);While(@pr)[(@bd);(@in);];];.

----

This expression defines a macro that allows for looping. MathPiper has a For function already, but this is how it could be defined in one line (In MathPiper the For function is bodied, we left that out here for clarity, as the example is about macros).

----

To see it work just type for(i:=0,i<3,i:=i+1,Echo(i)). You will see the count from one to three.

----

The construct works as follows; The expression defining the macro sets up a macro named for with four arguments. On the right is the body of the macro. This body contains expressions of the form @var. These are replaced by the values passed in on calling the macro. After all the variables have been replaced, the resulting expression is evaluated.

----

In effect a new program has been created. Such macro constructs come from LISP, and are famous for allowing you to almost design your own programming language constructs just for your own problem at hand. When used right, macros can greatly simplify the task of writing a program.

----

You can also use the back-quote ` to expand a macro in-place. It takes on the form `(expression), where the expression can again contain sub-expressions of the form @variable. These instances will be replaced with the values of these variables.

----

----

## Defining your own operators

Large part of the MathPiper system is defined in the scripting language itself. This includes the definitions of the operators it accepts, and their precedences. This means that you too can define your own operators. This section shows you how to do that.

----

Suppose we wanted to define a function F(x,y)=x/y+y/x. We could use the standard syntax F(a,b) := a/b + b/a;. F(1,2);.

----

For the purpose of this demonstration, lets assume that we want to define an infix operator xx for this operation.

----

We can teach MathPiper about this infix operator with Infix("xx", OpPrecedence("/"));. Here we told MathPiper that the operator xx is to have the same precedence as the division operator.

----

We can now proceed to tell MathPiper how to evaluate expressions involving the operator xx by defining it as we would with a function, a xx b := a/b + b/a;.

----

You can verify for yourself 3 xx 2 + 1; and 1 + 3 xx 2; return the same value, and that they follow the precedence rules (eg. xx binds stronger than +).

----

We have chosen the name xx just to show that we don't need to use the special characters in the infix operator's name. However we must define this operator as infix before using it in expressions, otherwise MathPiper will raise a syntax error.

----

Finally, we might decide to be completely flexible with this important function and also define it as a mathematical operator ## . First we define ## as a bodied function and then proceed as before. First we can tell MathPiper that ## is a bodied operator with Bodied("##", OpPrecedence("/"));. Then we define the function itself: ##(a) b := a xx b;. And now we can use the function, ##(1) 3 + 2;.

----

We have used the name ## but we could have used any other name such as xx or F or even _-+@+-_. Apart from possibly confusing yourself, it doesn't matter what you call the functions you define.

----

There is currently one limitation in MathPiper: once a function name is declared as infix (prefix, postfix) or bodied, it will always be interpreted that way. If we declare a function f to be bodied, we may later define different functions named f with different numbers of arguments, however all of these functions must be bodied.

----

When you use infix operators and either a prefix of postfix operator next to it you can run in to a situation where MathPiper can not quite figure out what you typed. This happens when the operators are right next to each other and all consist of symbols (and could thus in principle form a single operator). MathPiper will raise an error in that case. This can be avoided by inserting spaces.

----

One use of lists is the associative list, sometimes called a dictionary in other programming languages, which is implemented in MathPiper simply as a list of key-value pairs. Keys must be strings and values may be any objects.

----

Associative lists can also work as mini-databases, where a name is associated to an object.

----

As an example, first enter record:={}; to set up an empty record. After that, we can fill arbitrary fields in this record:

* record["name"]:="Isaia";
* record["occupation"]:="prophet";
* record["is alive"]:=False;

----

Now, evaluating record["name"] should result in the answer "Isaia". The record is now a list that contains three sublists, as you can see by evaluating record.

----

Assigning multiple values using lists.

Assignment of multiple variables is also possible using lists. For instance, evaluating {x,y}:={2!,3!} will result in 2 being assigned to x and 6 to y.

----

----

When assigning variables, the right hand side is evaluated before it is assigned. Thus a:=2*2 will set a to 4. This is however not the case for functions.

----

When entering f(x):=x+x the right hand side, x+x, is not evaluated before being assigned. This can be forced by using Eval().

----

Defining f(x) with f(x):=Eval(x+x) will tell the system to first evaluate x+x (which results in 2*x) before assigning it to the user function f.

----

This specific example is not a very useful one but it will come in handy when the operation being performed on the right hand side is expensive.

----

For example, if we evaluate a Taylor series expansion before assigning it to the user-defined function, the engine doesn't need to create the Taylor series expansion each time that user-defined function is called.

----

----

The imaginary unit i is denoted I and complex numbers can be entered as either expressions involving I, as for example 1+I*2, or explicitly as Complex(a,b) for a+ib. The form Complex(re,im) is the way MathPiper deals with complex numbers internally.

----

----

Linear Algebra

Vectors of fixed dimension are represented as lists of their components. The list {1, 2+x, 3*Sin(p)} would be a three-dimensional vector with components 1, 2+x and 3*Sin(p). Matrices are represented as a lists of lists.

----

Vector components can be assigned values just like list items, since they are in fact list items.

----

If we first set up a variable called "vector" to contain a three-dimensional vector with the command vector:=ZeroVector(3); (you can verify that it is indeed a vector with all components set to zero by evaluating vector), you can change elements of the vector just like you would the elements of a list (seeing as it is represented as a list).

----

For example, to set the second element to two, just evaluate vector[2] := 2;. This results in a new value for vector.

----

----

MathPiper can perform multiplication of matrices, vectors and numbers as usual in linear algebra. The standard MathPiper script library also includes taking the determinant and inverse of a matrix, finding eigenvectors and eigenvalues (in simple cases) and solving linear sets of equations, such as A * x = b where A is a matrix, and x and b are vectors.

----

As a little example to wetten your appetite, we define a Hilbert matrix: hilbert:=HilbertMatrix(3). We can then calculate the determinant with Determinant(hilbert), or the inverse with Inverse(hilbert). There are several more matrix operations supported. See the reference manual for more details.

----

----

"Threading" of functions

Some functions in MathPiper can be "threaded". This means that calling the function with a list as argument will result in a list with that function being called on each item in the list. E.g. Sin({a,b,c}); will result in {Sin(a),Sin(b),Sin(c)}.

----

This functionality is implemented for most normal analytic functions and arithmetic operators.

----

----

Functions as lists

For some work it pays to understand how things work under the hood. Internally, MathPiper represents all atomic expressions (numbers and variables) as strings and all compound expressions as lists, like LISP.

----

Try FullForm(a+b*c); and you will see the text (+ a (* b c )) appear on the screen. This function is occasionally useful, for example when trying to figure out why a specific transformation rule does not work on a specific expression.

----

If you try FullForm(1+2) you will see that the result is not quite what we intended. The system first adds up one and two, and then shows the tree structure of the end result, which is a simple number 3.

----

To stop MathPiper from evaluating something, you can use the function Hold, as FullForm(Hold(1+2)).

----

The function Eval is the opposite, it instructs MathPiper to re-evaluate its argument (effectively evaluating it twice). This undoes the effect of Hold, as for example Eval(Hold(1+2)).

----

----

Also, any expression can be converted to a list by the function Listify or back to an expression by the function UnList:

* Listify(a+b*(c+d));
* UnList({Atom("+"),x,1});

----

Note that the first element of the list is the name of the function +Atom("+") and that the subexpression b*(c+d) was not converted to list form. Listify just took the top node of the expression.

----

====

Example problems:

----

```
%yacas,output="latex"
    /* This is a great example problem to use in MathPiper.
    1) Enter expression.
    2) If it is a complicated expression, view it in LaTeX form to make
sure it has been entered correctly.  Use "Hold" around the expression to
make sure it is not evaluated and thus changed into another form.  In this
problem, if parentheses are not placed around the exponents then then the
    expression is evaluated differently than if they are present.
    3) Adjust the expression until it is correct.
    */

    a :=Hold(((1-x^(2*k))/(1-x))*((1-x^(2*(k+1)))/(1-x)));
    Write(a);

    %hoteqn
```

```
      $\frac{\left( 1 - x ^{2 \left( k + 1\right) }\right)  \left( 1 - x
^{2 k}\right) }{\left( 1 - x\right)  ^{2}} $
    %end

%end
----

%yacas,output="latex"
/*Be very careful to make sure all variables are in the intended
case.  Even one variable in the wrong case will make an expression's
meaning
different.
*/

    a := Hold( 1/2 * k *(k+1)+(k+1) );
    b := Hold( 1/2 *(k+1)*(k+2) );
    Write(TestMathPiper(a,b));

    %hoteqn
      $\mathrm{ True }$

        %output,preserve="false"
          HotEqn updated.
        %end

    %end

%end

----
%yacas,output=""
//Good example problem for newbies book.  From problem 19 in "Mathematical
Reasoning".
a(k) := (k+2)/(2*k+2);
b(k) := ( ((k+1)/(2*k)) * (1-(1/(k+1)^2) ) );
c(k) := (k+1)/(2*k) - (k+1)/(2*k*(k+1)^2);
d(k) := (k^3+3*k^2+2*k)/(2*k^3+4*k^2+2*k);
e(k) := (k^2+3*k+2)/(2*k^2+4*k+2);


//Write(d(k));
Write(TestMathPiper(a(k),e(k)));
//Write(Together(c(k)));
//Write(Simplify(c(k)));
//Write(Factor(Numer(Together(c(k)))):Factor(Denom(Together(c(k)))));

    %output,preserve="false"
      True
    %end
```

```
%end

====
----
Strings are generally represented with quotes around them, e.g. "this is a
string". Backslash \ in a string will unconditionally add the next
character to the string, so a quote can be added with \" (a backslash-quote
sequence).
----
1.3 Object types
MathPiper supports two basic kinds of objects: atoms and compounds. Atoms
are (integer or real, arbitrary-precision) numbers such as 2.71828,
symbolic variables such as A3 and character strings. Compounds include
functions and expressions, e.g. Cos(a-b) and lists, e.g. {1+a,2+b,3+c}.
The type of an object is returned by the built-in function Type, for
example:

In> Type(a);
Result> "";
In> Type(F(x));
Result> "F";
In> Type(x+y);
Result> "+";
In> Type({1,2,3});
Result> "List";
Internally, atoms are stored as strings and compounds as lists. (The
MathPiper lexical analyzer is case-sensitive, so List and list are
different atoms.) The functions String() and Atom() convert between atoms
and strings. A MathPiper list {1,2,3} is internally a list (List 1 2 3)
which is the same as a function call List(1,2,3) and for this reason the
"type" of a list is the string "List". During evaluation, atoms can be
interpreted as numbers, or as variables that may be bound to some value,
while compounds are interpreted as function calls.
Note that atoms that result from an Atom() call may be invalid and never
evaluate to anything. For example, Atom(3X) is an atom with string
representation "3X" but with no other properties.
Currently, no other lowest-level objects are provided by the core engine
besides numbers, atoms, strings, and lists. There is, however, a
possibility to link some externally compiled code that will provide
additional types of objects. Those will be available in MathPiper as
"generic objects." For example, fixed-size arrays are implemented in this
way.
----
Evaluation of an object is performed either explicitly by the built-in
command Eval() or implicitly when assigning variables or calling functions
with the object as argument (except when a function does not evaluate that
argument). Evaluation of an object can be explicitly inhibited using
Hold(). To make a function not evaluate one of its arguments, a
HoldArg(funcname, argname) must be declared for that function.
====
```

More from Google’s Calculator
▪ 100!/99!= ▪ 100!/99!=100
▪ 170!/169!= ▪ 170!/169!=170
▪ 171!/170!= ▪ <random search stuff>

POLS fails: why?
▪ The maximum “IEEE double float” number
1.7976931348623…◊ 10308 is a consequence
of arithmetic performance on most computers.
This particular computer-geeky limit has no
mathematical importance, but it means:
▪ 170! = 7.25741562... ◊ 10306 is smaller than this
and is legal.
▪ 171! is 1.241018070217…◊ 10309 which is
“too big.”
====
-5^2 evaluates to -25.  (-5)^2 evaluates to 25.
====
Describe how tabbing selected text moves it.
====
Describe inserting folds from the context menu.
====
Functions to cover:
!
Arbitrary math functions.
Append() - List operations.
Assoc()
Bin()
Ceil()
Count()
Concat()
Contains()
Delete() - List operations.
Difference() - Sets.
Divisors()
Drop() - List operations.
Div()
Exp()
Expand()
ExpandBrackets()
Factorize()
Factors()
FillList() - List operations.
Find()
Floor()
For()
ForEach()
FromBase()
Function()
Gcd()

GetTime()
Infinity
    Insert() - List operations.
Intersection() - Sets
Lcm()
Length()
Load()
Local()
Map()
MapSingle()
Max()
Min()
    Mod()
NewLine()
NextPrime()
Nl()
    Nth() - List operations.
NthRoot()
    Partition() - List operations.
Permutations()
ProperDivisors()
Random()
    RandomIntegerVector()
RandomSeed()
Rationalize()
    RemoveDuplicates() - List operations.
    Replace().
    Reverse() - List operations.
Round()
Sign()
    Select()
Space()
Sqrt()
Sum()
Table()
    Take() - List operations.
Time()
ToBase()
Undefined
Union() - Sets.
Until()
x--
x++