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

Sherm Ostrowsky

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 mathematics computing environment for performing numeric and symbolic computations (the difference between numeric and symbolic computations are discussed in a later section). Mathematics computing environments are complex and it takes a significant amount of time and effort to become proficient at using one. The amount of power that these environments make 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 Mathematics Computing Environment?

A Mathematics Computing Environment is a set of computer programs that 1) automatically execute 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 objects they create (An algorithm is a step-by-step sequence of instructions for solving a problem and we will be learning about algorithms later in the book).

Standard and graphing scientific calculator users interact with these devices using buttons and a small LCD display. In contrast to this, users interact with MathRider 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. Software that contains these kind of algorithms is commonly referred to as "Computer Algebra Systems (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 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 they desire and they can also enhance it.

2.2 What Is MathRider?

MathRider is an open source Mathematics Computing Environment 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.

(Note: If you already have a version of MathRider installed and you want to install a new version in the same directory that holds the old version, you must delete the old version first or move it to a separate directory.)

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. **(Note: be careful not to double click on the archive file by mistake when you are trying to open the mathrider folder. The Windows operating system will open the archive just like it opens folders and this can fool you into thinking you are opening the mathrider folder when you are not. You may want to move the archive file to another place on your hard drive after it has been extracted to avoid this potential confusion.)**

3.2.2 Extracting The Archive File For Unix Users

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

tar -xvjf <name of archive file>

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

3.3 MathRider's Directory Structure & Execution Instructions

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

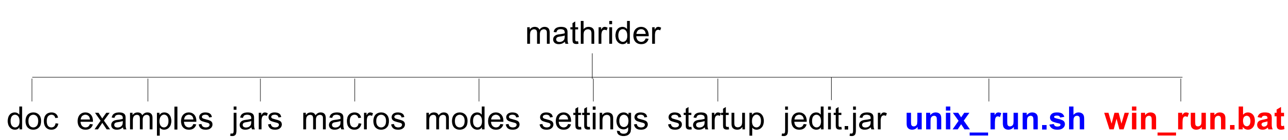

Illustration 1: MathRider's Directory Structure

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

doc - Contains MathRider's documentation files.

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

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

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

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

settings - Contains the application's main settings files.

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

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

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

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

3.3.1 Executing MathRider On Windows Systems

Open the **mathrider** folder **(not the archive file!)** 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. Programmer's text editors are similar to standard text editors (like NotePad and WordPad) and word processors (like MS Word and OpenOffice) in a number of ways so getting started with MathRider should be relatively easy for anyone who has used a text editor or a word processor. However, programmer's text editors are more challenging to use than a standard text editor or a word processor because programmer's text editors have capabilities that are far more advanced than these two types of applications.

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.

As you read through the following sections, I encourage you to explore each part of MathRider that is being discussed using your own copy of MathRider.

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, Folding, and View

These are advanced menus and they are described in later sections.

4.3.5 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.

4.3.6 Macros

This is an advanced menu and it is described in a later sections.

4.3.7 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. The tabs on the right side of the application which are labeled "GeoGebra", "Jung', "MathPiper", "MathPiperDocs", etc. are all plugins and they can be **opened** and **closed** by clicking on their **tabs**. **Feel free to close any of these plugins which may be opened if you are not currently using them.** MathRider pPlugins are covered in more depth in a later section.

4.3.8 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 MathPiper: A Computer Algebra System For Beginners

Computer algebra systems are extremely powerful and very useful for solving STEM-related problems. In fact, one of the reasons for creating MathRider was to provide a vehicle for delivering a computer 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 a CAS called 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.

5.1 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).

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 text-only **shell** or **command line** interface to it. This means that you type on the keyboard to send information to the console and it prints text to send you information.

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

Open the Console plugin by selecting the **Console** tab in the lower left part of the MathRider application. A text area will appear and in the upper left corner of this text area will be a pull down menu which may be set to "MathPiper" or to "System". If it is set to System, select this pull down menu and then select the **MathPiper** menu item that is inside of it. Feel free to increase the size of the console text area if you would like by dragging on the dotted lines which are at the top side and right side of the console window.

When the MathPiper console is first launched, it prints a welcome message and then provides **In>** as an input prompt:

```
MathPiper version ".75a".

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 result in **numeric form** can be obtained by using the **N() function**:

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

As can be seen here, when a result is given in numeric form, it means that it is given as a decimal number.

5.2.1 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.

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

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

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

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 is contained in the MathPiperDocs plugin and more of these functions will be discussed soon.**

5.2.2 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
```

5.2.3 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.

5.3 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 denominator of $\frac{5}{6}$ because it only holds the result of the most recent calculation and $\frac{5}{6}$ was calculated two steps back.

5.3.1 Variables

What would be nice is if MathPiper provided a way to store **results** (which are also called **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 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.

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.

5.3.1.1 Calculating With Unbound Variables

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.

5.3.1.2 Variable And Function Names Are Case Sensitive

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':

```
In> Box := 1
Result> 1

In> box := 2
Result> 2

In> Box
Result> 1

In> box
Result> 2
```

5.3.1.3 Using More Than One Variable

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.

Now that you have seen how to use the MathPiper console as both a **symbolic** and a **numeric** calculator, our next step is to take a closer look at the functions which are included with MathPiper. As you will soon discover, MathPiper contains an amazing number of functions which deal with a wide range of mathematics.

5.4 Exercises

Use the MathPiper console which is at the bottom of the MathRider application to complete the following exercises.

5.4.1 Exercise 1

Perform the following calculation:

$$\frac{1}{4}+\frac{3}{8}-\frac{7}{16}$$

5.4.2 Exercise 2

a) Assign the variable **ans** to the result of the calculation $\frac{1}{5}+\frac{7}{4}+\frac{15}{16}$ using the following line of code:

```
In> ans := 1/5 + 7/4 + 15/16
```

b) Use the Numer() function to calculate the numerator of ans.

c) Use the Denom() function to calculate the denominator of ans.

d) Use the N() function to calculate the numeric value of ans.

e) Use the Clear() function to unbind the variable ans and verify that ans is unbound by executing the following code and by using the IsBound() function:

```
In> ans
```

5.4.3 Exercise 3

Assign $\frac{1}{4}$ to variable **x**, $\frac{3}{8}$ to variable **y**, and $\frac{7}{16}$ to variable **z** using the **:=** operator. Then perform the following calculations:

a)
```
In> x
```

b)
```
In> y
```

```
c)
In> z
```

```
d)
In> x + y
```

```
d)
In> x + z
```

```
e)
In> x + y + z
```

6 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 MathPiperDocs plugin is available in a tab called "MathPiperDocs" which is near the right side of the MathRider application. Click on this tab to open the plugin and click on it again to close it.

The left side of the MathPiperDocs 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.

6.1 Function List

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

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

6.2 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.

6.3 Exercises

6.3.1 Exercise 1

Locate the N(), IsEven(), IsOdd(), Clear(), IsBound(), Numer(), Denom(), and Factor() functions in the **All Functions** section of the MathPiperDocs plugin and read the information that is available on them.

6.3.2 Exercise 2

Locate the N(), IsEven(), IsOdd(), Clear(), IsBound(), Numer(), Denom(), and Factor() functions in the **User Functions** section of the MathPiperDocs plugin and list which section each function is contained in. Don't include the **Alphabetical** or **Built In** subsections in your search.

7 Using MathRider As A Programmer's Text Editor

We have covered 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 and productive people in the world and most of their work is done using a programmer's text editor (or something similar to one). Programmers have designed programmer's text editors to be super-tools which can help them maximize their personal productivity and these tools have all kinds of capabilities that most people would not even suspect they contained.

Even though this book only covers a small part of the editing capabilities that MathRider has, what is covered will enable the user to begin writing useful programs.

7.1 Creating, Opening, Saving, And Closing 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**.

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 the user 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**.

A file can be closed by selecting **File->Close** from the menu bar and it can be opened by selecting **File->Open**.

7.2 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**.

7.3 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 is 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, MathRider worksheet files have a **.mrw** extension and MathRider knows what colors to highlight the various parts of a .mrw file in.

7.4 Exercises

7.4.1 Exercise 1

Create a text file called **"my_text_file.txt"** and place a few sentences in it. Save the text file somewhere on your hard drive then close it. Now, open the text file again using **File->Open** and verify that what you typed is still in the file.

8 MathRider Worksheet Files

While MathRider's ability to execute code inside a console provides 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**.

8.1 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 it and pressing **<shift><Enter>**. A fold always begins with a **start tag**, which starts with a percent symbol (**%)** followed by the **name of the fold type** (like this: **%<foldtype>**). The end of a fold is marked by an **end tag** which looks like **%/<foldtype>**. The only difference between a fold's start tag and its end tag is that the end tag has a slash (/) after the %.

For example, here is a MathPiper fold which will print the result of **2 + 3** to the MathPiper console (**Note: the line numbers are not part of the program and the semicolon ';' which is at the end of the line of code is required**):

```
%mathpiper

2 + 3;

%/mathpiper
```

The **output** generated by a fold (called the **parent fold**) is wrapped in a **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:
3:2 + 3;
4:
5:%/mathpiper
6:
7:    %output,preserve="false"
8:      Result: 5
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.

8.1.1 The Description Attribute

Folds can also have what is called a "**description attribute**" placed after the start tag which describes what the fold contains. For example, the following %mathpiper fold has a description attribute which indicates that the fold adds two number together:

```
%mathpiper,description="Add two numbers together."

2 + 3;

%/mathpiper
```

The description attribute is added to the start tag of a fold by placing a comma after the fold's type name and then adding the text **description="<text>"** after the comma. (**Note: no spaces can be present before or after the comma (,) or the equals sign (=)**).

8.2 Automatically Inserting Folds & Removing Unpreserved Folds

Typing the the top and bottom fold lines (for example:

```
%mathpiper

%/mathpiper
```

can be tedious and MathRider has a way to automatically insert them. Place the cursor at the beginning of a blank 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 and at the top of this menu are items which read "**Insert MathPiper Fold**", "**Insert Group Fold**", etc. If you select one of these menu items, an empty code fold of the proper type will automatically be inserted into the .mrw file at the position of the cursor.

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

8.3 Exercises

A MathRider worksheet file called "**newbies_book_examples_1.mrw**" can be obtained from this website:

https://mathrider.dev.java.net/source/browse/mathrider/trunk/src/doc/newbies_book/examples/proposed/misc/newbies_book_examples_1.mrw

It contains a number of %mathpiper folds which contain code examples from the previous sections of this book. Notice that all of the lines of code have a semicolon (;) placed after them. The reason this is needed is explained in a later section.

Download this worksheet file to your computer from the section on this website that contains the highest revision number and then open it in MathRider. Then, use the worksheet to do the following exercises.

8.3.1 Exercise 1

Execute folds 1-8 in the top section of the worksheet by placing the cursor inside of the fold and then pressing <shift><enter> on the keyboard.

8.3.2 Exercise 2

The code in folds 9 and 10 have errors in them. Fix the errors and then execute the folds again.

8.3.3 Exercise 3

Use the empty fold 11 to calculate the expression 100 - 23;

8.3.4 Exercise 4

Perform the following calculations by creating new folds at the bottom of the worksheet (using the right-click popup menu) and placing each calculation into its own fold:

a) 2*7 + 3

b) 18/3

c) 234238342 + 2038408203

d) 324802984 * 2308098234

e) Factor the result which was calculated in d).

9 MathPiper Programming Fundamentals

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.

9.1 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
```

9.2 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 (which is also used as the "result of the last calculation" symbol), 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 (Note: there must be a space between the two negative signs):

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

In MathPiper, **operators** are symbols (or groups of symbols) which are implemented with **functions**. One can either call the function that 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.

9.3 Operator Precedence

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

^ Exponents are evaluated right to left.

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

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

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

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

And here it is in traditional form:

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

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**.

9.4 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
```

(Note: precedence adjusting parentheses are different from the parentheses that are used to call functions.)

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.

9.5 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 **IsEven()** 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**, or **methods**.

The functions that come with MathPiper have names which consist of either a single word (such as **Add()**) 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.

9.6 Functions That Produce Side Effects

Most functions are executed to obtain the **results** they produce but some functions are executed in order to **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 the computer's memory.

When a function produces a side effect which sends information to the user, this information has the words **Side Effects:** placed before it in the output instead of the word **Result:**. The **Echo()** and **Write()** functions are examples of functions that produce side effects and they are covered in the next section.

9.6.1 The Echo(), Write(), and NewLine() 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".

9.6.1.1 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 (separated by spaces) as side effect output. If it is desired that commas be printed between the numbers in the output, simply place three commas between the expressions that are passed to Echo():

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

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 last expression that was evaluated (which is usually 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 to 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
```

9.6.1.2 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 besides the one discussed here and more information about them can be found in the documentation for these functions.

9.6.1.3 NewLine()

The **NewLine()** function simply prints a blank line in the side effects output. It is useful for placing vertical space between printed lines:

```
%mathpiper

a := 1;
Echo(a);
NewLine();

b := 2;
Echo(b);
NewLine();

c := 3;
Echo(c);

%/mathpiper

    %output,preserve="false"
      Result: True

      Side Effects:
      1

      2

      3
    %/output
```

9.7 Expressions Are Separated By Semicolons

As discussed earlier, all of the expressions that are inside of a **%mathpiper** fold must have a semicolon (;) after them. However, 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.

9.7.1 Placing More Than One Expression On A Line In A Fold

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 remove the spaces from the previous code, the output remains the same:

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

9.7.2 Placing More Than One Expression On A Line In The Console Using A Code Block

The MathPiper console is only able to execute one expression at a time so if the previous code that executes three variable assignments and three Echo() functions on a single line is evaluated in the console, only the expression **a := 1** is executed:

```
In> a := 1; Echo(a); b := 2; Echo(b); c := 3; Echo(c);
Result> 1
```

Fortunately, this limitation can be overcome by placing the code into a **code block**. A **code block** (which is also called a **compound expression**) consists of one or more expressions which are separated by semicolons and placed within an open bracket (**[**) and close bracket (**]**) pair. If a code block is evaluated in the MathPiper console, each expression in the block will be executed from left to right. The following example shows the previous code being executed within of a code block inside the MathPiper console:

```
In> [a := 1; Echo(a); b := 2; Echo(b); c := 3; Echo(c);]
Result> True
Side Effects>
1
2
3
```

Notice that this time all of the expressions were executed and 1-3 was printed as a side effect. Code blocks always return the result of the last expression executed as the result of the whole block. In this case, True was returned as the result because the last Echo(c) function returned True. If we place another expression after the Echo(c) function, however, the block will execute this new expression last and its result will be the one returned by the block:

```
In> [a := 1; Echo(a); b := 2; Echo(b); c := 3; Echo(c); 2 + 2]
Result> 4
Side Effects>
1
2
3
```

Code blocks are very powerful and we will be discussing them further in later sections.

9.8 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
```

9.8.1 The MathPiper Console and MathPiper Folds Can Access The Same Variables

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."
```

9.8.2 Using Strings To Make Echo's Output Easier To Read

When the Echo() function is used to print the values of multiple variables, it is often helpful to print some information next to each variable so that it is easier to determine which value came from which Echo() function in the code. The following program prints the name of the variable that each value came from next to it in the side effects output:

```
%mathpiper

a := 1;
Echo("Variable a: ", a);

b := 2;
```

```
Echo("Variable b: ", b);

c := 3;
Echo("Variable c: ", c);

%/mathpiper

    %output,preserve="false"
      Result: True

      Side Effects:
      Variable a: 1
      Variable b: 2
      Variable c: 3
    %/output
```

9.8.3 Accessing The Individual Letters In A String

Individual letters in a string (which are also called **characters**) can be accessed by placing the character's position number (also called an **index**) inside of brackets **[]** after the variable it is bound to. A character's position is determined by its distance from the left side of the string starting at 1. For example, in the string "Hello", '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 := "Hello, I am a string."
Result> "Hello, I am a string."

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"
```

9.9 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. Therefore, comments are ignored by the computer when a 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
```

9.10 Exercises

For the following exercises, create a new MathRider worksheet file called **section_9_exercises_<your first name>_<your last name>.mrw**. (**Note: there are no spaces in this file name**). For example, John Smith's worksheet would be called:

section_9_exercises_john_smith.mrw.

After this worksheet has been created, place your answer for each exercise into its own fold in this worksheet. Place a description attribute in the start tag of each fold which indicates the exercise the fold contains the solution to. The folds you create should look similar to this one:

```
%mathpiper,description="Exercise 1"

//Sample fold.

%/mathpiper
```

9.10.1 Exercise 1

Change the precedence of the following expression using parentheses so that it prints 20 instead of 14:

```
2 + 3 * 4
```

9.10.2 Exercise 2

Place the following calculations into a fold and then use one Echo() function per variable to print the results of the calculations. Put strings in the Echo() functions which indicate which variable each calculated value is bound to:

```
a := 1+2+3+4+5;
b := 1-2-3-4-5;
c := 1*2*3*4*5;
d := 1/2/3/4/5;
```

9.10.3 Exercise 3

Place the following calculations into a fold and then use one Echo() function to print the results of all the calculations on a single line (Remember, the Echo() function can print multiple values if they are separated by commas.):

```
Clear(x);
a := 2*2*2*2*2;
b := 2^5;
```

```
c := x^2 * x^3;
d := 2^2 * 2^3;
```

9.10.4 Exercise 4

The following code assigns a string which contains all of the upper case letters of the alphabet to the variable **upper**. Each of the three Echo() functions prints an index number and the letter that is at that position in the string. Place this code into a fold and then continue the Echo() functions so that all 26 letters and their index numbers are printed

```
upper := "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

Echo(1,upper[1]);
Echo(2,upper[2]);
Echo(3,upper[3]);
```

9.10.5 Exercise 5

Use Echo() functions to print an index number and the character at this position for the following string (this is similar to what was done in Exercise 4.):

```
extra := ".!@#$%^&*() _+<>,?/{}[]|\-=";

Echo(1,extra[1]);
Echo(2,extra[2]);
Echo(3,extra[3]);
```

9.10.6 Exercise 6

The following program uses strings and index numbers to print a person's name. Create a program which uses the three strings from this program to print the names of three of your favorite movie actors.

```
%mathpiper
/*
  This program uses strings and index numbers to print
  a person's name.
*/

upper := "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
lower := "abcdefghijklmnopqrstuvwxyz";
extra := ".!@#$%^&*() _+<>,?/{}[]|\-=";


//Print "Mary Smith.".
Echo(upper[13],lower[1],lower[18],lower[25],extra[12],upper[19],lower[13],l
ower[9],lower[20],lower[8],extra[1]);


%/mathpiper
```

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

      Side Effects:
      Mary Smith.
.   %/output
```

10 Rectangular Selection Mode And Text Area Splitting

10.1 Rectangular Selection Mode

One capability that MathRider has that a word processor 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.**

One thing that rectangular selection mode is very handy for is removing the line numbers from folds that are copied from this document into a MathRider worksheet. If you have been manually removing these line numbers with the arrow, delete, and backspace keys, I think you will find that using rectangular selection mode to remove them is much easier.

10.2 Text area splitting

Sometimes it is useful to have two or more text areas open for a single document or multiple documents so that different parts of the documents can be edited at the same time. A situation where this would have been helpful was in the previous section where the output from an exercise in a MathRider worksheet contained a list of index numbers and letters which was useful for completing a later exercise.

MathRider has this ability and it is called **splitting**. If you look just to the right of the toolbar there is an icon which looks like a blank window, an icon to the right of it which looks like a window which was split horizontally, and an icon to the right of the horizontal one which is split vertically. If you let your mouse hover over these icons, a short description will be displayed for each of them. **(For now, ignore the icon which has a yellow sunburst on it. It is the New View icon and it is an advanced feature.)**

Select a text area and then experiment with splitting it by pressing the horizontal and vertical splitting buttons. Move around these split text areas with their scroll bars and when you want to unsplit the document, just press the "**Unsplit All**" icon.

11 Working With Random Integers

It is often useful to use random integers in a program. For example, a program may need to simulate the rolling of dice in a game. In this section, a function for obtaining nonnegative integers is discussed along with how to use it to simulate the rolling of dice.

11.1 Obtaining Nonnegative Random Integers With The RandomInteger() Function

One way that a MathPiper program can generate random integers is with the RandomInteger() function **(Note: the RandomInteger() function is not currently listed in the MathPiperDocs plugin.)** The RandomInteger() function takes an integer as a parameter and it returns a random integer between 0 and the passed in integer. The following example shows random integers between 0 and 4 **inclusive** being obtained from RandomInteger(). **Inclusive** here means that both 0 and 4 are included in the range of random integers that may be returned. If the word **exclusive** was used instead, this would mean that neither 0 nor 4 would be in the range.

```
In> RandomInteger(5)
Result> 3
In> RandomInteger(5)
Result> 4
In> RandomInteger(5)
Result> 3
In> RandomInteger(5)
Result> 1
In> RandomInteger(5)
Result> 2
In> RandomInteger(5)
Result> 4
In> RandomInteger(5)
Result> 1
In> RandomInteger(5)
Result> 1
In> RandomInteger(5)
Result> 0
In> RandomInteger(5)
Result> 1
```

If generating integers between 1 and 5 inclusive is desired instead of integers between 0 an 4, the number 1 can simply be added to the value which is returned by RandomInteger():

```
In> RandomInteger(5) + 1
```

```
Result> 4
In> RandomInteger(5) + 1
Result> 5
In> RandomInteger(5) + 1
Result> 4
In> RandomInteger(5) + 1
Result> 2
In> RandomInteger(5) + 1
Result> 3
In> RandomInteger(5) + 1
Result> 5
In> RandomInteger(5) + 1
Result> 2
In> RandomInteger(5) + 1
Result> 2
In> RandomInteger(5) + 1
Result> 1
In> RandomInteger(5) + 1
Result> 2
```

Random integers between 1 and 100 can be generated by passing 100 to RandomInteger() and adding 1 to the result.:

```
In> RandomInteger(100) + 1
Result> 15
In> RandomInteger(100) + 1
Result> 14
In> RandomInteger(100) + 1
Result> 82
In> RandomInteger(100) + 1
Result> 93
In> RandomInteger(100) + 1
Result> 32
```

A range of random integers that does not start with 0 or 1 can also be generated. For example, random integers between 50 and 100 can be obtained by having RandomInteger() generate a random integer between 0 and 49 and then adding 1 and 50 to the result:

```
In> RandomInteger(50) + 1 + 50
Result> 87
In> RandomInteger(50) + 1 + 50
Result> 100
In> RandomInteger(50) + 1 + 50
Result> 76
In> RandomInteger(50) + 1 + 50
Result> 60
In> RandomInteger(50) + 1 + 50
Result> 73
```

11.2 Simulating The Rolling Of Dice

The following example shows the simulated rolling of a single six sided die using the RandomInteger() function:

```
In> RandomInteger(6) + 1
Result> 5
In> RandomInteger(6) + 1
Result> 6
In> RandomInteger(6) + 1
Result> 3
In> RandomInteger(6) + 1
Result> 2
In> RandomInteger(6) + 1
Result> 5
```

Code that simulates the rolling of two 6 sided dice can be evaluated in the MathPiper console by placing it within a **code block**. The following code outputs the sum of the two simulated dice:

```
In> [a := RandomInteger(6) + 1; b := RandomInteger(6) + 1; a + b;]
Result> 6
In> [a := RandomInteger(6) + 1; b := RandomInteger(6) + 1; a + b;]
Result> 12
In> [a := RandomInteger(6) + 1; b := RandomInteger(6) + 1; a + b;]
Result> 6
In> [a := RandomInteger(6) + 1; b := RandomInteger(6) + 1; a + b;]
Result> 4
In> [a := RandomInteger(6) + 1; b := RandomInteger(6) + 1; a + b;]
Result> 3
In> [a := RandomInteger(6) + 1; b := RandomInteger(6) + 1; a + b;]
Result> 8
```

Now that we have the ability to simulate the rolling of two 6 sided dice, it would be interesting to determine if some sums of these dice occur more frequently than other sums. What we would like to do is to roll these simulated dice hundreds (or even thousands) of times and then analyze the sums that were produced. We don't have the programming capability to easily do this yet, but after we finish the section on while loops, we will.

12 Making Decisions

The simple programs that have been discussed up to this point show some of the power that software makes available to programmers. However, these programs are limited in their problem solving ability because they are unable to make decisions. This section shows how programs which have the ability to make decisions are able to solve a wider range of problems than programs that can't make decisions.

12.1 Conditional Operators

A program's decision making ability is based on a set of special operators which are called **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 only 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

This example shows some of these conditional operators being evaluated in the MathPiper console:

```
In> 1 < 2
Result> True

In> 4 > 5
Result> False
```

```
In> 8 >= 8
Result> True

In> 5 <= 10
Result> True
```

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.

12.2 Predicate Expressions

Expressions which return either **True** or **False** are called "**predicate**" expressions. By themselves, predicate expressions are not very useful and they only become so when they are used with special decision making functions, like the If() function (which is discussed in the next section).

12.3 Exercises

12.3.1 Exercise 1

Open a MathPiper session and evaluate the following predicate expressions:

```
In> 3 = 3

In> 3 = 4

In> 3 < 4

In> 3 != 4

In> -3 < 4

In> 4 >= 4

In> 1/2 < 1/4

In> 15/23 < 122/189

/*In the following two expressions, notice that 1/2 is not considered to be
equal to .5 unless it is converted to a numerical value first.*/

In> 1/2  = .5

In> N(1/2) = .5
```

12.3.2 Exercise 2

Come up with 10 predicate expressions of your own and evaluate them in the MathPiper console.

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

All programming languages have 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)
```

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. (Note: any function that accepts a predicate expression as a parameter can also accept the boolean values True and False).

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 **first Echo()** function. The **second Echo()** function at the bottom of the program prints "End of program" regardless of what the If() function does. (**Note: semicolons cannot be placed after expressions which are in function calls.**)

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.

12.4.1 If() Functions Which Include An "Else" Parameter

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
```

12.5 Exercises

12.5.1 Exercise 1

Write a program which uses the RandomInteger() function to simulate the flipping of a coin (Hint: you can use 1 to represent a head and 0 to represent a tail.). Use predicate expressions, the If() function, and the Echo() function to print the string **"The coin came up heads."** or the string **"The coin came up tails."**, depending on what the simulated coin flip came up as when the code was executed.

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

12.6.1 And()

Sometimes a programmer 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** (or **notations**) and this is the first one:

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

This calling format is able to accept one or more predicate 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 return a **False**, then the And() function will return a **False**. This can be seen in the following example:

```
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 also demonstrates the infix version of the And() function being used:

```
%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
```

12.6.2 Or()

The Or() function is similar to the And() function in that it has both a function calling format and an infix calling format and it only works with predicate expressions. 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 this example shows Or() being used with function calling 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 this example shows infix 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 the infix version of the Or() function being used:

```
%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
```

12.6.3 Not() & Prefix Notation

The **Not()** function works with predicate 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()'s function calling format:

```
Not(expression)
```

and this example shows Not() being used with function calling 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 also 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
```

12.7 Exercises

12.7.1 Exercise 1

The following program simulates the rolling of two dice and prints a message if **both** of the two dice come up less than or equal to 3. Create a similar program which simulates the flipping of two coins and print the message "Both coins came up heads." if both coins come up heads.

```
%mathpiper
/*
  This program simulates the rolling of two dice and prints a message if
  both of the two dice come up less than or equal to 3.
*/

dice1 := RandomInteger(6) + 1;
dice2 := RandomInteger(6) + 1;

Echo("Dice1: ", dice1, "  Dice2: ", dice2);
NewLine();

If( dice1 <= 3 And dice2 <= 3, Echo("Both dice came up <= to 3.") );

%/mathpiper
```

12.7.2 Exercise 2

The following program simulates the rolling of two dice and prints a message if **either** of the two dice come up less than or equal to 3. Create a similar program which simulates the flipping of two coins and print the message "At least one coin came up heads." if at least one coin comes up heads.

```
%mathpiper
/*
  This program simulates the rolling of two dice and prints a message if
  either of the two dice come up less than or equal to 3.
*/

dice1 := RandomInteger(6) + 1;
dice2 := RandomInteger(6) + 1;

Echo("Dice1: ", dice1, "  Dice2: ", dice2);
NewLine();

If( dice1 <= 3 Or dice2 <= 3, Echo("At least one die came up <= 3.") );

%/mathpiper
```

13 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 MathPiper 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 expressions it contains as long as its "predicate" expression is **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 a **code block** (code blocks were discussed in an earlier section). What a function's body is will become clearer after studying some example programs.

13.1 Printing The Integers From 1 to 10

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 predicate 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 indeed less than or equal to 10, a value of **True** is returned by the predicate expression.

The While() function sees that the predicate 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 the body of the 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**.

13.2 Printing The Integers From 1 to 100

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
```

13.3 Printing The Odd Integers From 1 To 99

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
```

13.4 Printing The Integers From 1 To 100 In Reverse Order

Finally, the following program prints the integers 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.

13.5 Expressions Inside Of Code Blocks Are Indented

In the programs in the previous sections which use while loops, notice that the expressions which are inside of the While() function's code block are **indented**. These expressions do not need to be indented to execute properly, but doing so makes the program easier to read.

13.6 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 opening the MathPiper console and then pressing the "**Stop**" button which it contains. The Stop button is circular and it has an X on it. (**Note: currently this button only works if MathPiper is executed inside of a %mathpiper fold.**)

Lets experiment with the **Stop** 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; //Oops, x is not being incremented!.
];

%/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 still running the code.

Execute this program now and then interrupt it using the "**Stop**" button. When the program is interrupted, the %output fold will display the message "**User interrupted calculation**" to indicate that the program was interrupted. After a program has been interrupted, the program can be edited and then rerun.

13.7 A Program That Simulates Rolling Two Dice 50 Times

The following program is larger than the previous programs that have been discussed in this book, but it is also more interesting and more useful. It uses a While() loop to simulate the rolling of two dice 50 times and it records how many times each possible sum has been rolled so that this data can be printed. The comments in the code explain what each part of the program does. (Remember, if you copy this program to a MathRider worksheet, you can use **rectangular selection mode** to easily remove the line numbers).

```
%mathpiper
/*
  This program simulates rolling two dice 50 times.
*/


/*
  These variables are used to record how many times
  a possible sum of two dice has been rolled.  They are
  all initialized to 0 before the simulation begins.
*/
numberOfTwosRolled := 0;
numberOfThreesRolled := 0;
numberOfFoursRolled := 0;
numberOfFivesRolled := 0;
numberOfSixesRolled := 0;
numberOfSevensRolled := 0;
numberOfEightsRolled := 0;
numberOfNinesRolled := 0;
numberOfTensRolled := 0;
numberOfElevensRolled := 0;
numberOfTwelvesRolled := 0;


//This variable keeps track of the number of the current roll.
roll := 1;


Echo("These are the rolls:");


/*
 The simulation is performed inside of this while loop.  The number of
 times the dice will be rolled can be changed by changing the number 50
 which is in the While function's predicate expression.
*/
While(roll <= 50)
[
    //Roll the dice.
    die1 := RandomInteger(6) + 1;
    die2 := RandomInteger(6) + 1;
```

```


    //Calculate the sum of the two dice.
    rollSum := die1 + die2;


    /*
     Print the sum that was rolled.  Note: if a large number of rolls
     is going to be performed (say > 1000), it would be best to comment
     out this Write() function so that it does not put too much text
     into the output fold.
    */
    Write(rollSum,,);


    /*
     These If() functions determine which sum was rolled and then add
     1 to the variable which is keeping track of the number of times
     that sum was rolled.
    */
    If(rollSum = 2, numberOfTwosRolled := numberOfTwosRolled + 1);
    If(rollSum = 3, numberOfThreesRolled := numberOfThreesRolled + 1);
    If(rollSum = 4, numberOfFoursRolled := numberOfFoursRolled + 1);
    If(rollSum = 5, numberOfFivesRolled := numberOfFivesRolled + 1);
    If(rollSum = 6, numberOfSixesRolled := numberOfSixesRolled + 1);
    If(rollSum = 7, numberOfSevensRolled := numberOfSevensRolled + 1);
    If(rollSum = 8, numberOfEightsRolled := numberOfEightsRolled + 1);
    If(rollSum = 9, numberOfNinesRolled := numberOfNinesRolled + 1);
    If(rollSum = 10, numberOfTensRolled := numberOfTensRolled + 1);
    If(rollSum = 11, numberOfElevensRolled := numberOfElevensRolled+1);
    If(rollSum = 12, numberOfTwelvesRolled := numberOfTwelvesRolled+1);


    //Increment the roll variable to the next roll number.
    roll := roll + 1;
];


//Print the contents of the sum count variables for visual analysis.
NewLine();
NewLine();
Echo("Number of Twos rolled: ", numberOfTwosRolled);
Echo("Number of Threes rolled: ", numberOfThreesRolled);
Echo("Number of Fours rolled: ", numberOfFoursRolled);
Echo("Number of Fives rolled: ", numberOfFivesRolled);
Echo("Number of Sixes rolled: ", numberOfSixesRolled);
Echo("Number of Sevens rolled: ", numberOfSevensRolled);
Echo("Number of Eights rolled: ", numberOfEightsRolled);
Echo("Number of Nines rolled: ", numberOfNinesRolled);
Echo("Number of Tens rolled: ", numberOfTensRolled);
Echo("Number of Elevens rolled: ", numberOfElevensRolled);
Echo("Number of Twelves rolled: ", numberOfTwelvesRolled);
```

```


%/mathpiper

    %output,preserve="false"
      Result: True

      Side effects:
      These are the rolls:
      4,8,6,4,6,9,7,11,9,3,11,6,11,7,11,4,7,7,8,7,3,6,7,7,7,12,4,
     12,7,8,12,6,8,10,10,5,9,8,4,5,3,5,7,7,4,6,7,7,5,8,

      Number of Twos rolled: 0
      Number of Threes rolled: 3
      Number of Fours rolled: 6
      Number of Fives rolled: 4
      Number of Sixes rolled: 6
      Number of Sevens rolled: 13
      Number of Eights rolled: 6
      Number of Nines rolled: 3
      Number of Tens rolled: 2
      Number of Elevens rolled: 4
      Number of Twelves rolled: 3
.   %/output
```

13.8 Exercises

13.8.1 Exercise 1

Create a program which uses a while loop to print the even integers from 2 to 50 inclusive.

13.8.2 Exercise 2

Create a program which prints all the multiples of 5 between 5 and 50 inclusive.

13.8.3 Exercise 3

Create a program which simulates the flipping of a coin 500 times. Print the number of times the coin came up heads and the number of times it came up tails after the loop is finished executing.

14 Predicate Functions

A **predicate function** is a function that either returns **True** or **False**. Most predicate functions in MathPiper have names which 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
```

The complete list of predicate functions is contained in the **User Functions/Predicates** node in the MathPiperDocs plugin.

14.1 Finding Prime Numbers With A Loop

Predicate functions are very powerful when they are combined with loops because they can be used to automatically make numerous checks. The following program uses a while loop to pass the integers 1 through 20 (one at a time) to the **IsPrime()** function in order to determine which integers are prime and which integers are not prime:

```
%mathpiper

//Determine which numbers between 1 and 20 (inclusive) are prime.

x := 1;

While(x <= 20)
[
    primeStatus := IsPrime(x);

    Echo(x, "is prime: ", primeStatus);

    x := x + 1;
];

%/mathpiper

    %output,preserve="false"
      Result: True

      Side Effects:
      1 is prime: False
      2 is prime: True
      3 is prime: True
      4 is prime: False
      5 is prime: True
      6 is prime: False
      7 is prime: True
      8 is prime: False
      9 is prime: False
      10 is prime: False
      11 is prime: True
      12 is prime: False
```

```
        13 is prime: True
        14 is prime: False
        15 is prime: False
        16 is prime: False
        17 is prime: True
        18 is prime: False
        19 is prime: True
        20 is prime: False
.    %/output
```

This program worked fairly well, but it is limited because it prints a line for each prime number and also each non-prime number. This means that if large ranges of integers were processed, enormous amounts of output would be produced. The following program solves this problem by using an If() function to only print a number if it is prime:

```
%mathpiper

//Print the prime numbers between 1 and 50 (inclusive).

x := 1;

While(x <= 50)
[
    primeStatus := IsPrime(x);

    If(primeStatus = True, Write(x,,) );

    x := x + 1;

];

%/mathpiper

    %output,preserve="false"
      Result: True

      Side Effects:
      2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,
.    %/output
```

This program is able to process a much larger range of numbers than the previous one without having its output fill up the text area. However, the program itself can be shortened by moving the **IsPrime()** function **inside** of the **If()** function instead of using the **primeStatus** variable to communicate with it:

```
%mathpiper

/*
```

```
    Print the prime numbers between 1 and 50 (inclusive).
    This is a shorter version which places the IsPrime() function
    inside of the If() function instead of using a variable.
*/

x := 1;

While(x <= 50)
[

    If(IsPrime(x), Write(x,,) );

    x := x + 1;

];

%/mathpiper

    %output,preserve="false"
      Result: True

      Side Effects:
      2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,
.   %/output
```

14.2 Finding The Length Of A String With The Length() Function

Strings can contain zero or more characters and the **Length()** function can be used to determine how many characters a string holds:

```
In> s := "Red"
Result> "Red"

In> Length(s)
Result> 3
```

In this example, the string "Red" is assigned to the variable **s** and then **s** is passed to the **Length()** function. The **Length()** function returned a **3** which means the string contained **3 characters**.

The following example shows that strings can also be passed to functions directly:

```
In> Length("Red")
Result> 3
```

An **empty string** is represented by **two double quote marks with no space in between them**. The **length** of an empty string is **0**:

```
In> Length("")
Result> 0
```

14.3 Converting Numbers To Strings With The String() Function

Sometimes it is useful to convert a number to a string so that the individual digits in the number can be analyzed or manipulated. The following example shows a **number** being converted to a **string** with the **String()** function so that its **leftmost** and **rightmost** digits can be placed into **variables**:

```
In> number := 523
Result> 523

In> stringNumber := String(number)
Result> "523"

In> leftmostDigit := stringNumber[1]
Result> "5"

In> rightmostDigit := stringNumber[ Length(stringNumber) ]
Result> "3"
```

Notice that the Length() function is used here to determine which character in **stringNumber** held the **rightmost** digit.

14.4 Finding Prime Numbers Which End With 7 (And Multi-line Function Calls)

Now that we have covered how to turn a number into a string, lets use this ability inside a loop. The following program finds all the **prime numbers** between **1** and **500** which have a **7 as their rightmost digit**. There are three important things which are shown in this program:

1) Function calls **can have their parameters placed on more than one line** if the parameters are too long to fit on a **single line**. In this case, a long code block is being placed inside of an If() function.

2) Code blocks (which are considered to be compound expressions) **cannot have a semicolon placed after them if they are in a function call**. If a semicolon is placed after this code block, an error will be produced.

3) If() functions can be placed inside of other If() functions in order to make more complex decisions. This is referred to as **nesting** functions.

When the program is executed, it finds 24 prime numbers which have 7 as their rightmost digit:

```
%mathpiper

/*
    Find all the prime numbers between 1 and 500 which have a 7
    as their rightmost digit.
*/

x := 1;

While(x <= 500)
[
    //Notice how function parameters can be put on more than one line.
    If(IsPrime(x),
        [
            stringVersionOfNumber := String(x);

            stringLength := Length(stringVersionOfNumber);

            //Notice that If() functions can be placed inside of other
            // If() functions.
            If(stringVersionOfNumber[stringLength] = "7", Write(x,,) );

        ] //Notice that semicolons cannot be placed after code blocks
          //which are in function calls.

    ); //This is the close parentheses for the outer If() function.

    x := x + 1;
];

%/mathpiper

    %output,preserve="false"
      Result: True

      Side Effects:
      7,17,37,47,67,97,107,127,137,157,167,197,227,257,277,307,317,
      337,347,367,397,457,467,487,
.   %/output
```

It would be nice if we had the ability to store these numbers someplace so that they could be processed further and this is discussed in the next section.

14.5 Exercises

14.5.1 Exercise 1

Write a program which uses a loop to determine how many prime numbers there are between 1 and 1000. You do not need to print the numbers themselves, just how many there are.

14.5.2 Exercise 2

Write a program which uses a loop to print all of the prime numbers between 10 and 99 which contain the digit 3 in either their 1's place, or their 10's place, or both places.

15 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 be made to **grow and shrink as needed**, and they can be **nested**. Expressions in a list can be **accessed by their position** in the list (similar to the way that characters in a string are accessed) and they can also be **replaced by other expressions**.

One way to create a list is by placing zero or more expressions separated by commas inside of a **pair of braces {}**. In the following example, a list is created that contains various expressions and then it is assigned 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 that is bound to the list and then putting the expression's position number inside of the brackets (**Note: the first expression in the list is at position 1 counting from the left end 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 an **unbound 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 integers **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.

15.1 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, **Append()** 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 new list bound to the variable that is 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 new 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 they are advanced functions which are not covered in this book.

15.2 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 number in the list.

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

While(y <= Length(x))
[
    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 <= Length(testList))
[
    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**.

15.2.1 Using A While Loop And Append() To Place Values In A List

In an earlier section it was mentioned that it would be nice if we could store a set of values for later processing and this can be done with a **while loop** and the

Append() function. The following program creates an empty list and assigned it to the variable **primes**. The **while loop** and the **IsPrime()** function is then used to locate the prime integers between 1 and 50 and the **Append()** function is used to place them in the list. The last part of the program then prints some information about the numbers that were placed into the list:

```
%mathpiper

//Place the prime numbers between 1 and 50 (inclusive) into a list.

//Create an empty list.
primes := {};

x := 1;

While(x <= 50)
[
    /*
        If x is prime, append it to the end of the list and then assign
        the new list that is created to the variable 'primes'.
    */
    If(IsPrime(x), primes := Append(primes, x ) );

    x := x + 1;
];

//Print information about the primes that were found.
Echo("Primes ", primes);
Echo("The number of primes in the list = ", Length(primes) );
Echo("The first number in the list = ", primes[1] );

%/mathpiper

    %output,preserve="false"
      Result: True

      Side Effects:
      Primes {2,3,5,7,11,13,17,19,23,29,31,37,41,43,47}
      The number of primes in the list = 15
      The first number in the list = 2
.   %/output
```

The ability to place values into a list with a loop is very powerful and we will be using this ability throughout the rest of the book.

15.3 Exercises

15.3.1 Exercise 1

```
Create a program that uses a loop and an IsOdd() function to analyze the
following list and then print the number of odd numbers it contains.

{73,94,80,37,56,94,40,21,7,24,15,14,82,93,32,74,22,68,65,52,85,61,46,86,25}
```

15.3.2 Exercise 2

Create a program that uses a loop and an IsNegativeNumber() function to copy all of the negative numbers in the following list into a new list. Use the variable **negativeNumbers** to hold the new list.

```
{36,-29,-33,-6,14,7,-16,-3,-14,37,-38,-8,-45,-21,-26,6,6,38,-20,33,41,-
4,24,37,40,29}
```

15.3.3 Exercise 3

Create a program that uses a loop to analyze the following list and then print the following information about it:

1) The largest number in the list.
2) The smallest number in the list.
3) The sum of all the numbers in the list.

```
{73,94,80,37,56,94,40,21,7,24,15,14,82,93,32,74,22,68,65,52,85,61,46,86,25}
```

15.4 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() also uses bodied notation like the While() function 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**.

15.5 Print All The Values In A List Using A ForEach() function

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
```

15.6 Calculate The Sum Of The Numbers In A List Using ForEach()

In previous examples, counting code in the form **x := x + 1** was used to count how many times a while loop was executed. The following program uses a **ForEach()** function and a line of code similar to this counter to calculate the **sum of the numbers in a list**:

```
%mathpiper

/*
  This program calculates the sum of the numbers
  in a list.
*/

//This variable is used to accumulate the sum.
sum := 0;

ForEach(x, {1,2,3,4,5,6,7,8,9,10} )
[
    /*
      Add the contents of x to the contents of sum
      and place the result back into sum.
    */
    sum := sum + x;
```

```

    //Print the sum as it is being accumulated.
    Write(sum,,);
];

NewLine(); NewLine();

Echo("The sum of the numbers in the list = ", sum);

%/mathpiper

    %output,preserve="false"
      Result: True

      Side Effects:
      1,3,6,10,15,21,28,36,45,55,

      The sum of the numbers in the list = 55
.   %/output
```

In the above program, the integers **1** through **10** were manually placed into a list by typing them individually. This method is limited because only a relatively small number of integers can be placed into a list this way. The following section discusses an operator which can be used to automatically place a large number of integers into a list with very little typing.

15.7 The .. Range Operator

```
first .. last
```

A programmer often needs to create a list which contains **consecutive integers** and the **..** "**range"** operator can be used to do this. The **first** integer in the list is placed before the **..** operator and the **last** integer in the list is placed after it (**Note: there must be a space immediately to the left of the .. operator and a space immediately to the right of it or an error will be generated.**). 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> 1 .. 500
Result> {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}

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 these examples show, the .. operator can generate lists of integers in ascending order and descending order. It can also generate lists that are very large and ones that contain negative integers.

Remember, though, if one or both of the spaces around the .. are omitted, an error is generated:

```
In> 1..3
Result>
Error parsing expression, near token .3.
```

15.8 Using ForEach() With The Range Operator To Print The Prime Numbers Between 1 And 100

The following program shows how to use a **ForEach()** function instead of a **While()** function to print the prime numbers between 1 and 100. Notice that loops that are implemented with **ForEach() often require less typing** than their **While()** based equivalents:

```
%mathpiper

/*
  This program prints the prime integers between 1 and 100 using
  a ForEach() function instead of a While() function.  Notice that
  the ForEach() version requires less typing than the While()
  version.
*/

ForEach(x, 1 .. 100)
[
    If(IsPrime(x), Write(x,,) );
];

%/mathpiper

    %output,preserve="false"
      Result: True

      Side Effects:
      2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,
      73,79,83,89,97,
.   %/output
```

15.8.1 Using ForEach() And The Range Operator To Place The Prime Numbers Between 1 And 50 Into A List

A ForEach() function can also be used to place values in a list, just the the While() function can:

```
%mathpiper

/*
 Place the prime numbers between 1 and 50 into
 a list using a ForEach() function.
*/

//Create a new list.
primes := {};

ForEach(number, 1 .. 50)
[
    /*
      If number is prime, append it to the end of the list and
      then assign the new list that is created to the variable
      'primes'.
    */
    If(IsPrime(number), primes := Append(primes, number ) );
];

//Print information about the primes that were found.
Echo("Primes ", primes);
Echo("The number of primes in the list = ", Length(primes) );
Echo("The first number in the list = ", primes[1] );

%/mathpiper

    %output,preserve="false"
      Result: True

      Side Effects:
      Primes {2,3,5,7,11,13,17,19,23,29,31,37,41,43,47}
      The number of primes in the list = 15
      The first number in the list = 2
.   %/output
```

As can be seen from the above examples, the **ForEach()** function and the **range operator** can do a significant amount of work with very little typing. You will discover in the next section that MathPiper has functions which are even more powerful than these two.

15.8.2 Exercises

15.8.3 Exercise 1

Create a program that uses a **ForEach()** function and an **IsOdd()** function to analyze the following list and then print the number of odd numbers it contains.

```
{73,94,80,37,56,94,40,21,7,24,15,14,82,93,32,74,22,68,65,52,85,61,46,86,25}
```

15.8.4 Exercise 2

Create a program that uses a **ForEach()** function and an **IsNegativeNumber()** function to copy all of the negative numbers in the following list into a new list. Use the variable **negativeNumbers** to hold the new list.

```
{36,-29,-33,-6,14,7,-16,-3,-14,37,-38,-8,-45,-21,-26,6,6,38,-20,33,41,-
4,24,37,40,29}
```

15.8.5 Exercise 3

Create a program that uses a **ForEach()** function to analyze the following list and then print the following information about it:

1) The largest number in the list.
2) The smallest number in the list.
3) The sum of all the numbers in the list.

```
{73,94,80,37,56,94,40,21,7,24,15,14,82,93,32,74,22,68,65,52,85,61,46,86,25}
```

15.8.6 Exercise 4

Create a program that uses a **while loop** to make a list that contains **1000 random integers** between **1** and **100** inclusive. Then, use a **ForEach()** function to determine how many integers in the list are **prime** and use an **Echo()** function to print this total.

16 Functions & Operators Which Loop Internally

Looping is such a useful capability that MathPiper has many functions which loop internally. Now that you have some experience with loops, you can use this experience to help you imagine how these functions use loops to process the information that is passed to them.

16.1 Functions & Operators Which Loop Internally To Process Lists

This section discusses a number of functions that use loops to process lists.

16.1.1 TableForm()

```
TableForm(list)
```

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

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

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

16.1.2 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 to the opposite truth value:

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

16.1.3 Find()

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

The **Find()** function searches a list for the first occurrence of a given expression. If the expression is found, the **position of 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
```

16.1.4 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
```

16.1.5 Select()

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

Select() returns a list that contains all the expressions in a list which make a given predicate function 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.

16.1.6 The Nth() Function & The [] Operator

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

The **Nth()** function simply returns the expression which is at a given position 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
```

16.1.7 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}
```

16.1.8 Concat()

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

The Concat() function is short for "concatenate" which means to join together sequentially. It 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}
```

16.1.9 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}
```

16.1.10 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}
```

16.1.11 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}
```

16.1.12 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}
```

16.1.13 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}
```

16.1.14 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}
```

16.1.15 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> Length(testList) % 3
Result> 2
```

Remember that **%** is the remainder operator. It divides two integers and returns their remainder.

16.1.16 Table()

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

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

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

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

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

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

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

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

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

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

16.2 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.

16.2.1 RandomIntegerVector()

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

A vector is 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}
```

16.2.2 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
```

16.2.3 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
```

16.2.4 Gcd()

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

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

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

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

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

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

16.2.5 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
```

16.2.6 Add()

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

Add() can find the sum of two or more values that are 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
```

16.2.7 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
```

16.3 Exercises

16.3.1 Exercise 1

Create a program that uses **RandomIntegerVector()** to create a 100 member list that contains random integers between 1 and 5 inclusive. Use **Count()** to determine how many of each digit 1-5 are in the list and then print this information.

16.3.2 Exercise 2

Create a program that uses **RandomIntegerVector()** to create a 100 member list that contains random integers between 1 and 50 inclusive and use **Contains()** to determine if the number 25 is in the list. Print "25 was in the list." if 25 was found in the list and "25 was not in the list." if it wasn't found.

16.3.3 Exercise 3

Create a program that uses **RandomIntegerVector()** to create a 100 member list that contains random integers between 1 and 50 inclusive and use Find**()** to determine if the number 10 is in the list. Print the position of 10 if it was found in the list and "10 was not in the list." if it wasn't found.

16.3.4 Exercise 4

Create a program that uses **RandomIntegerVector()** to create a 100 member list that contains random integers between 0 and 3 inclusive. Use **Select()** with the **IsNonZeroInteger()** predicate function to obtain all of the nonzero integers in this list.

16.3.5 Exercise 5

Create a program that uses **Table()** to obtain a list which contains the squares of the integers between 1 and 10 inclusive.

17 Nested Loops

Now that you have seen how to solve problems with single loops, it is time to discuss what can be done when a loop is placed inside of another loop. A loop that is placed **inside** of another loop it is called a **nested loop** and this nesting can be extended to numerous levels if needed. This means that loop 1 can have loop 2 placed inside of it, loop 2 can have loop 3 placed inside of it, loop 3 can have loop 4 placed inside of it, and so on.

Nesting loops allows the programmer to accomplish an enormous amount of work with very little typing.

17.1 Generate All The Combinations That Can Be Entered Into A Two Digit Wheel Lock Using Two Nested Loops

The following program generates all the combinations that can be entered into a two digit wheel lock. It uses a nested loop to accomplish this with the "**inside**" nested loop being used to generate **one's place** digits and the "**outside**" loop being used to generate **ten's place** digits.

```
%mathpiper

/*
  Generate all the combinations can be entered into a two
  digit wheel lock.
*/

combinations := {};

ForEach(digit1, 0 .. 9) //This loop is called the "outside" loop.
```

```
[
    ForEach(digit2, 0 .. 9)//This loop is called the "inside" loop.
    [
        combinations := Append(combinations, {digit1, digit2});
    ];
];

Echo(TableForm(combinations));

%/mathpiper

    %output,preserve="false"
      Result: True

      Side Effects:
      {0,0}
      {0,1}
      {0,2}
      {0,3}
      {0,4}
      {0,5}
      {0,6}
        .
        .  //The middle of the list has not been shown.
        .
       {9,3}
       {9,4}
       {9,5}
       {9,6}
       {9,7}
       {9,8}
       {9,9}
       True
.   %/output
```

The relationship between the outside loop and the inside loop is interesting because each time the **outside loop cycles once**, the **inside loop cycles 10 times**. Study this program carefully because nested loops can be used to solve a wide range of problems and therefore understanding how they work is important.

17.2 Exercises

17.2.1 Exercise 1

Create a program that will generate all of the combinations that can be entered into a three digit wheel lock. (Hint: a triple nested loop can be used to accomplish this.)

18 User Defined Functions

In computer programming, a **function** is a named section of code that can be **called** from other sections of code. **Values** can be sent to a function for processing as part of the **call** and a function always returns a value as its result. A function can also generate side effects when it is called and side effects have been covered in earlier sections.

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

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

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

This line of code defined a new function called **addNums** and specified that it will accept two values when it is called. The **first** value will be placed into the variable **num1** and the **second** value will be placed into the variable **num2**.

Variables like num1 and num2 which are used in a function to accept values from calling code are called **formal parameters**. **Formal parameter variables** are used inside a function to process the **values/actual parameters/arguments** that were placed into them by the calling code.

The code on the **right side** of the **assignment operator** is **bound** to the function name "**addNums**" and it is executed each time **addNums()** is called. The following example shows the new **addNums()** function being called multiple times with different values being passed to it:

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

In> addNums(4,5)
Result> 9

In> addNums(9,1)
Result> 10
```

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

defined function names to begin with a lower case letter to distinguish them from the functions that come with MathPiper.

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

```
%mathpiper

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

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

    resultList;
];

%/mathpiper

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

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

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

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

18.1 Global Variables, Local Variables, & Local()

The new **evenIntegers()** function seems to work well, but there is a problem.

The variables '**x**' and **resultList** were defined inside the function as **global variables** which means they are accessible from anywhere, including from within other functions, within other folds (as shown here):

```
%mathpiper

Echo(x, ",", resultList);

%/mathpiper

    %output,preserve="false"
      Result: True

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

and from within the MathPiper console:

```
In> x
Result> 12

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

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

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

```
%mathpiper

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

evenIntegers(endInteger) :=
[
```

```
    Local(x,resultList);

    resultList := {};
    x := 2;

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

    resultList;
];

%/mathpiper

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

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

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

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

In> x
Result> x

In> resultList
Result> resultList
```

18.2 Exercises

18.2.1 Exercise 1

Create a function called **tenOddIntegers()** which returns a list which contains 10 random odd integers between 1 and 99 inclusive.

18.2.2 Exercise 2

Create a function called **convertStringToList(string)** which takes a string as a parameter and returns a list which contains all of the characters in the string. Here is an example of how the function should work:

```
In> convertStringToList("Hello friend!")
Result> {"H","e","l","l","o"," ","f","r","i","e","n","d","!"}
```

```
In> convertStringToList("Computer Algebra System")
Result> {"C","o","m","p","u","t","e","r"," ","A","l","g","e","b","r","a","
","S","y","s","t","e","m"}
```

THE INFORMATION BELOW THIS LINE IS STILL IN DEVELOPMENT

19 Models

Up to this point in the book, a significant number of software-based tools have been discussed and in this section you will start to learn how to use these tools to do useful work. Doing useful work is the main reason that computers were invented. However, before one can understand how to useful work with a computer, one must first understand what a **model** is.

One example of a model that most people are familiar with is a scaled-down plastic model car. When a scaled-down version of an object is made it means that a smaller copy of the object is created, with each of the dimensions of all of its parts being shrunken by the same amount. For example, if a scaled-down car was 50 times smaller than a given full-size car, then all of the parts in the scaled-down car would be 50 times smaller than their analogous parts in the full-size car.

Of course, the model car does not contain small working copies of all of the parts of a real car because it would be very difficult to create small working copies of all of the parts in a real car. It probably could be done, but it would be very expensive. This is why models are usually used to **represent** objects instead of either scaled or unscaled exact **copies** of the objects. A **model** is a simplified **representation** of an object that only represents **some** of its attributes. Examples of typical object attributes include weight, height, strength, and color.

The attributes that are selected for representation 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**.

As an example, suppose we wanted to build a garage that could hold two 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 two 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."

If you are thinking "Thats ridiculous, 11 garages would need to be built using this strategy instead of just one." you are correct. A way to solve the 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, maybe 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.

What makes these pieces of paper models of the full-size objects they represent and not exact scaled-down copies of them is that the only attributes of the full-sized objects that were represented by pieces of paper were the object's length and width. The process of placing only some of an object's attributes into a model instead of all of them is called abstraction.

19.1 Placing Models Into A Computer

Now that we have discussed what a model is you may find it interesting to know that the reason one of the first modern programmable digital computers was invented was to model the paths of artillery projectiles. Its name was ENIAC and it was was invented in the 1940s. This shows that the process of modeling with computers is one of the fundamental reasons that people use computers and using computers as modeling tools has continued to the present day.

As discussed in the previous section, simple tools like paper, scissors, and a pencil can be used to create models, but tools like these are very limited in their modeling abilities

"I can see how paper can be used to model things...