Introduction To Programming With MathPiper And MathPiperIDE

by Ted Kosan

Table of Contents

1 Preface

1.1 Dedication

This book is dedicated to Steve Yegge and his blog entries "Math Every Day" (http://steve.yegge.googlepages.com/math-every-day) and "Math For Programmers" (http://steve-yegge.blogspot.com/2006/03/math-for-programmers.html).

1.2 Website And Support Email List

The website for MathPiper and MathPiperIDE is http://mathpiper.org.

The support email list for this book is called **mathpiper-user@googlegroups.com** and you can subscribe to it at http://groups.google.com/group/mathpiper-user

1.3 Recommended Weekly Sequence When Teaching A Class With This Book

- Week 1: Sections 1 - 7.
- Week 2: Sections 8 - 9.
- Week 3: Sections 10 - 13.
- Week 4: Sections 14 - 15.
- Week 5: Sections 16 - 19.
- Week 6: Exam

2 Introduction

MathPiperIDE 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 MathPiperIDE, but fortunately one does not need to be a MathPiperIDE 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 that 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 MathPiperIDE using a rich graphical user interface that is driven by a computer keyboard and mouse. Almost any personal computer can be used to run MathPiperIDE, 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 that 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 that have access to the mathematics algorithms that 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 (which is discussed in a later section) 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 MathPiperIDE?

MathPiperIDE is an open source Mathematics Computing Environment that 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.

MathPiperIDE uses MathPiper as its default computer algebra system, BeanShell as its main scripting language, jEdit as its development environment, and Java as its overall implementation language. One way to determine a person's MathPiperIDE expertise is by their knowledge of these components. (see Table 1)

Level	Knowledge
MathPiperIDE Developer	Knows Java, BeanShell, and JEdit at an advanced level. Is able to develop MathPiperIDE plugins.
MathPiperIDE Customizer	Knows Java, BeanShell, and JEdit at an intermediate level. Is able to develop MathPiperIDE macros.
MathPiperIDE Expert	Knows MathPiper at an advanced level and is skilled at using most aspects of the MathPiperIDE application.
MathPiperIDE Novice	Knows MathPiper at an intermediate level, but has only used MathPiperIDE for a short while.
MathPiperIDE Beginner	Does not know MathPiper but has been exposed to at least one programming language.
Programming Beginner	Does not know how a computer works and has never programmed before but knows how to use a word processor.

Table 1: MathPiperIDE user experience levels.

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

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

2.3 What Inspired The Creation Of MathPiperIDE?

One of the main inspirations for MathPiper is 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) 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

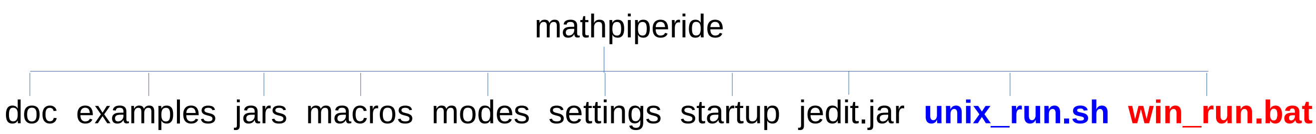
3 Downloading, Installing, And Executing MathPiperIDE

Instructions for downloading and installing MathPiperIDE are on the download page of the MathPiper website (http://mathpiper.org).

3.1 MathPiperIDE's Directory Structure

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

Illustration 1: MathPiperIDE's Directory Structure



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

doc - Contains MathPiperIDE's documentation files.

examples - Contains various example programs, some of which are pre-opened when MathPiperIDE 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 that tell MathPiperIDE 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 MathPiperIDE launches.

jedit.jar - Holds the core jEdit application that MathPiperIDE builds upon.

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

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

4 The Graphical User Interface

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

In the following sections, the main parts of MathPiperIDE'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 MathPiperIDE that is being discussed using your own copy of MathPiperIDE.

4.1 Buffers And Text Areas

In MathPiperIDE, 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 that displays the name of the buffer it is working on along with an indicator that 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 MathPiperIDE'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 that 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 that 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 that 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 that contains access to most of the search actions.

At the top of this dialog window is a text area labeled **Search for** that allows the user to enter text they would like to find. Immediately below it is a text area labeled **Replace with** that is for entering optional text that can be used to replace text that 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 that 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 MathPiperIDE 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 that are labeled "JFreeChart”, "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.** MathPiperIDE pPlugins are covered in more depth in a later section.

4.3.8 Help

The most important action in the **Help** menu is the **MathPiperIDE Help** action. This action brings up a dialog window with contains documentation for the core MathPiperIDE 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 that 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.

4.4.1 Undo And Redo

The **Undo** button on the toolbar is able to undo any text was entered since the current session of MathPiperIDE was launched. This is very handy for undoing mistakes or getting back text that was deleted. The **Redo** button can be used if you have selected Undo too many times and you need to "undo" one ore more Undo operations.

5 Using MathPiperIDE As A Programmer's Text Editor

We have covered some of MathPiperIDE's mathematics capabilities and this section discusses some of its programming capabilities. As indicated in a previous section, MathPiperIDE 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 that 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 MathPiperIDE has, what is covered will enable the user to begin writing useful programs.

5.1 Creating, Opening, Saving, And Closing Text Files

A good way to begin learning how to use MathPiperIDE'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, MathPiperIDE 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**.

5.2 Editing Files

If you know how to use a word processor, then it should be fairly easy for you to learn how to use MathPiperIDE 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**.

5.3 File Modes

Text file names are suppose to have a file extension that 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.).

MathPiperIDE uses a file's extension type to place its text area into a customized **mode** that highlights various parts of its contents. For example, MathPiperIDE worksheet files have a **.mpws** extension and MathPiperIDE knows what colors to highlight the various parts of a .mpws file in.

5.4 Learning How To Type Properly Is An Excellent Investment Of Your Time

This is a good place in the document to mention that learning how to type properly is an investment that will pay back dividends throughout your whole life. Almost any work you do on a computer (including programming) will be done *much* faster and with less errors if you know how to type properly. Here is what Steve Yegge has to say about this subject:

"If you are a programmer, or an IT professional working with computers in *any* capacity, **you need to learn to type!** I don't know how to put it any more clearly than that."

A good way to learn how to type is to locate a free "learn how to type" program on the web and use it.

5.5 Exercises

5.5.1 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.`

6 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 MathPiperIDE 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 that is included by default with MathPiperIDE.

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.

6.1 Numeric Vs. Symbolic Computations

A Computer Algebra System (CAS) is software that 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 MathPiperIDE, 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.

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

Open the MathPiperConsole plugin by selecting the **MathPiperConsole** tab in the lower left part of the MathPiperIDE application. The MathPiper **console** interface is a text area that is inside this plugin. The size of the console text area can be changed by dragging on the dotted lines that 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 "xxx".

In>
```

Click to the right of the prompt in order to place the cursor there then type **2+2** followed by **<enter> (or <return> on a Macintosh)**:

```
In> 2+2
Result: 4

In>
```

When **<enter>** was pressed, 2 + 2 was read into MathPiper for **evaluation** and **Result:** was printed followed by the result **4**. The numeral 4 is the **value** that was returned by **evaluating** 2 + 2. 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.

Previous input can be automatically entered to the right of an In> prompt by placing the cursor to the right of the prompt, pressing the <ctrl> key, and then pressing the up and down arrow keys.

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> 1/2
Result: 1/2
```

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 **NM() procedure** (which is discussed in the next section):

```
In> NM(1/2)
Result: 0.5
```

As can be seen here, when a result is given in numeric form, it means that it is given as a **decimal number**. A numeric result could also be obtained by using a decimal point either after the 1 or the 2 (or both of them):

```
In> 1./2
Result: 0.5

In> 1/2.
Result: 0.5

In> 1./2.
Result: 0.5
```

When one or more decimal numbers are used in a calculation, MathPiper will usually return a numeric result.

6.2.1 Using Procedures

NM() is an example of a **procedure**. A procedure can be thought of as a "black box" that accepts input, processes the input, and returns a result. Each procedure has a name and in this case, the name of the procedure is **NM**, which stands for **"Numeric Mode"**. To the right of a procedure's name there is always a **set of parentheses**, and information that is sent to the procedure is placed inside of them. The purpose of the **NM()** procedure is to make sure that the information that is sent to it is processed numerically instead of symbolically. Procedures are used by **evaluating** them, and this happens when <enter> is pressed. Another name for evaluating a procedure is **calling** it.

6.2.1.1 The Sqrt() Square Root Procedure

The following example show the **NM()** procedure being used with the square root procedure **Sqrt()**:

```
In> Sqrt(9)
Result: 3

In> Sqrt(8)
Result: Sqrt(8)

In> NM(Sqrt(8))
Result: 2.828427125
```

Notice that Sqrt(9) returned 3 as expected but Sqrt(8) returned Sqrt(8). We needed to use the NM() procedure to force the square root procedure to return a numeric result. The reason that Sqrt(8) does not appear to have done anything is because computer algebra systems like to work with expressions that are as exact as possible. In this case the **symbolic** value Sqrt(8) represents the number that is the square root of 8 more accurately than any decimal number can.

For example, the following four decimal numbers all represent $\sqrt{(8)}$, but none of them represent it more accurately than Sqrt(8) does:

2.828427125

2.82842712474619

2.828427124746190097603377448

2.82842712474619009760337744841939615713934375075

Whenever MathPiper returns a symbolic result and a numeric result is desired, simply use the NM() procedure to obtain one. The ability to work with symbolic values are one of the things that make computer algebra systems so powerful, and they are discussed in more depth in later sections.

6.2.1.2 The Even?() Procedure

An example of a simple procedure is **Even?()**. The **Even?()** procedure takes a number as input and returns **True** if the number is even and **False** if it is not even:

```
In> Even?(4)
Result: True

In> Even?(5)
Result: False
```

MathPiper has a large number of procedures, 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 procedures is contained in the MathPiperDocs plugin, and more of these procedures will be discussed soon.**

6.2.2 Accessing Previous Input And Results

The MathPiper console is like a mini text editor, which means you can copy text from it, paste text into it, and edit existing text. You can also reevaluate previous input by simply placing the cursor on the desired **In>** line and pressing **<enter>** on it again.

The console also keeps a history of all input lines that have been evaluated. If the cursor is placed on any **In>** line, pressing **<ctrl><up arrow>** will display each previous line of input that has been entered.

Finally, the MathPiperConsole associates the most recent computation result with the number sign (**#**) 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 (Note: there needs to be a space between the # and * characters.)
Result: 80
```

6.3 Saving And Restoring A Console Session

If you need to save or open the contents of a console session, you can use the "File" menu that is present in the upper left corner of the MathPiperConsole window.

6.3.1 Syntax Errors

An expression's **syntax** is related to whether it is **typed** correctly or not. If input is sent to MathPiper that 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

Exception: Error encountered during parsing: Error parsing expression near
token ***( \ )***. Starting at index 3
```

To fix this problem, change the \ to a /, and reevaluate the expression.

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

6.4 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, let's 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
```

Let's say that you would like to have MathPiper determine the numerator of this result. This can be done by using (or **calling**) the **Numerator()** procedure:

```
In> Numerator(#)
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.

6.4.1 Variables And The Variable State

What would be nice is if MathPiper provided a way to assign **results** (which are also called **values**) to symbols that we choose, instead of ones that it chooses. Fortunately, this is exactly what it does! **Names** 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 **index**.

Even though variable names can start with an upper case letter, by convention all variables should begin with a lower case letter. If the name is composed of more than one word, the first letter of each word after the first word should be capitalized as shown in these examples:

numberOfDoors, **seatsInRoom6**, and **averageTemperature**.

Note: the underscore (_) character cannot be used in variable names. One or more underscore characters in a name identify it as a constant. A constant is a name that always evaluates to itself, and it is discussed shortly.

The process of associating a value with a variable is called **assigning** 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 the **assignment operator** (**:=**) and an **expression** on the **right** side of this operator. This expression is evaluated, and the value it returns is **assigned** to the variable. For example, the following code assigns the value 5 to the variable 'a':

```
In> a := 5
Result: 5
```

The assignment operator (**:=**) is read as "**becomes**", and therefore the above expression reads " **'a' becomes 5**".

If the variable 'a' is evaluated by itself, it returns the value that is currently assigned to it:

```
In> a
Result: 5
```

The assignment operator (**:=**) is meant to look like an arrow that points from right to left like ← in order to emphasize the right-to-left assignment of variables.

Let's 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> Numerator(a)
Result: 5

In> Denominator(a)
Result: 6
```

In this example, the assignment operator (**:=**) was used to assign the result value $\frac{5}{6}$ to the variable 'a'. **When 'a' was evaluated by itself, the value that was most recently assigned to it (in this case $\frac{5}{6}$) was returned.** This value will stay assigned to the variable 'a' as long as MathPiper is running, unless 'a' is unassigned with the **Unassign()** procedure, 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 procedures in turn. **(Note: there can be no spaces between the : and the = in the := operator)**

6.4.1.1 The Global Variable State

The **global variable state** is the list of all of the global variables that are currently assigned, along with the values that have been assigned to them. A global variable is a variable that is accessible by all the code in the system. The other main kind of variable is a local variable. Local variables (which are covered in a later section) are accessible to limited sections of code. All variables that we will be using in the MathPiper console are global variables.

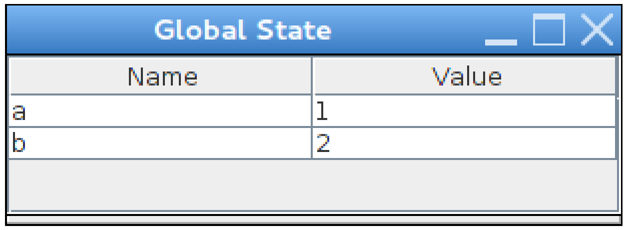
The **State()** procedure can be used to obtain a copy of the global variable state:

```
In> a := 1
Result: 1

In> b := 2
Result: 2

In> State()
Result: [a:1,b:2]
```

The **State** button in the console can also be used to view the global state. When this button is pressed, a window is shown that contains the global variable state:



It is a good idea to keep a current variable state window open while programming because it makes it easier to see what effects the code is producing.

6.4.1.2 Evaluating An Unassigned Variable Throws An Exception

If an unassigned variable is evaluated, an exception is thrown:

```
In> Unassign(a)
Result: True

In> a
Result: Exception
Exception: The variable <a> does not have a value assigned to it.
```

The Unassign() procedure unassigns a variable, and it returns the value **True** as a result to indicate that the variable that was sent to it was successfully **unassigned**. Many procedures return either return **True** or **False** to indicate whether or not the operation they performed succeeded. True and False are constants, and constants are discussed in the next section.

6.4.1.3 Constants

A **constant** is a name that evaluates to itself. The following is list of some constants that are predefined in MathPiper:

- True
- False
- Infinity
- Undefined
- All
- None

The constant **Infinity** evaluates to itself:

```
In> Infinity
Result: Infinity
```

If an attempt is made to assign a value to a constant, an exception is thrown:

```
In> Infinity := 3
Result: Exception
Exception: <Infinity> is a constant, and values cannot be assigned to
constants.
```

As mentioned earlier, some procedures return a predefined constant as a value. For example, the Assigned?() procedure returns **True** if a variable is assigned, and it returns **False** if it is unassigned:

```
In> a := 1
Result: 1

In> a
Result: 1

In> Assigned?(a)
Result: True
```

All currently assigned variables can be unassigned by passing the **constant 'All' to Unassign**:

```
In> b := 2
Result: 2

In> State()
Result: [a:1,b:2]

In> Unassign(All)
Result: True

In> State()
Result: []
```

One way to indicate that a name is a constant is to use an underscore character (_) as the first letter in the name:

_x, **_y**, **_heavy**

Constants that start with an underscore evaluate to themselves:

```
In> _x
Result: _x
```

and values cannot be assigned to these constants either:

```
In> _x := 3
Result: Exception
<_x> is a constant, and values cannot be assigned to constants.
```

Numbers are also constants because they evaluate to themselves:

```
In> 3
Result: 3
```

6.4.1.4 Calculating With Constants

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

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

In> Factor(14)
Result: 2*7

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

Now let's factor an expression that contains the constant '_x':

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

In> Expand(#)
Result: _x^2+24*_x+80
```

Factor() uses the rules of algebra to **manipulate** the algebraic expression that is sent to it into factored form. The **Expand()** procedure was then able to take the factored expression (_x+4)*(_x+20) and manipulate it until it was expanded. One way to remember what the procedures **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.

6.4.1.5 Variable And Constant Names Are Case Sensitive

MathPiper variable and constant names are **case sensitive**. This means 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
```

6.4.1.6 Using More Than One Variable

Programs are able to have more than one variable. The following example shows three variables being used:

```
In> a := 2
Result: 2

In> b := 3
Result: 3

In> a + b
Result: 5

In> answer := (a + b)
Result: 5

In> 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 value 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 procedures that are included with MathPiper. As you will soon discover, MathPiper contains numerous procedures that deal with a wide range of mathematics.

6.5 Exercises

Use the MathPiper console that is at the bottom of the MathPiperIDE application to complete the following exercises.

6.5.1 Exercise 1

Answer each one of the following questions:

a) What is the purpose of the NM() procedure?

b) What is a variable?

c) Are the variables 'x' and 'X' the same variable?

d) What is the difference between an assigned variable and an unassigned variable?

e) What happens if you evaluate an unassigned variable?

f) How can a value be assigned to a variable?

g) How can a variable be unassigned?

h) What does the # character do?

6.5.2 Exercise 2

Perform the following calculation:

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

6.5.3 Exercise 3

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

```
In> answer := (1/5 + 7/4 + 15/16)
```

b) Use the Numerator() procedure to calculate the numerator of **answer**.

c) Use the Denominator() procedure to calculate the denominator of **answer**.

d) Use the NM() procedure to calculate the numeric value of **answer**.

e) Use the Unassign() procedure to unassign the variable **answer** and verify that **answer** is unassigned by using the State() procedure and by using the Global State window.

6.5.4 Exercise 4

Assign $\frac{1}{4}$ to variable **x**, $\frac{3}{8}$ to variable **y**, and $\frac{7}{16}$ to variable **z** using the **:=** operator (**remember, there is no space between the : and the =**). Then perform the following calculations:

a)
```
In> x
```

b)
```
In> y
```

c)
```
In> z
```

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

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

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

7 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 MathPiperIDE 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 procedures that come with MathPiper and the right side of the window contains a mini-browser that can be used to navigate the documentation.

7.1 Procedure List

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

Both the user and programmer procedure names have been placed into a "tree" on the left side of the MathPiperDocs window to allow for easy navigation. The branches of the procedure 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 procedures they contain organized into categories and the **top category in each branch** lists all the procedures in the branch in **alphabetical order** for quick access. Clicking on a procedure 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 procedures that are in the procedure tree! Most MathPiperIDE beginners will not know what most of them mean, and some will not know what any of them mean. Part of the benefit MathPiperIDE provides is exposing the user to the existence of these categories and procedures. The more you use MathPiperIDE, the more you will learn about these categories and procedures and someday you may even get to the point where you understand all of them. This book is designed to show beginners how to begin using these procedures using a gentle step-by-step approach.

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

7.3 Exercises

7.3.1 Exercise 1

Locate the NM(), Even?(), Odd?(), Unassign(), Assigned?(), Numerator(), Denominator(), and State() procedures in the **All Procedures** section of the MathPiperDocs plugin, and read the information that is available on them. List the **one line descriptions** that are at the top of the documentation for each of these procedures.

7.3.2 Exercise 2

Locate the NM(), Even?(), Odd?(), Unassign(), Assigned?(), Numerator(), Denominator(), and State() procedures in the **Mathematical Procedures** section or the **Programming Procedures** section of the MathPiperDocs plugin and list which **category** each procedure is contained in. **Don't** include the **Alphabetical** or **Built In** categories in your search. For example, the **NM()** procedure is in the **Numbers (Operations)** category.

8 MathPiperIDE Worksheet Files

While MathPiperIDE's ability to execute code inside a console provides a significant amount of power to the user, most of MathPiperIDE's power is derived from **worksheets**. MathPiperIDE worksheets are text files that have a **.mpws** extension and are which are able to execute multiple types of code in a single text area. The **worksheet_demo_1.mpws** file (which is preloaded in the MathPiperIDE environment when it is first launched) demonstrates how a worksheet is able to execute multiple types of code in what are called **code folds**. (**Note: a new .mpws file needs to be saved immediately after it is created, because MathPiperIDE will not recognize it as a MathPiper worksheet until it has been saved.**)

8.1 Code Folds And Source Code

A code fold is a named section inside a MathPiperIDE worksheet that contains source code which can be executed by placing the cursor inside of it and pressing **<shift><Enter>**. One or more expressions that are typed into a code fold are called a **computer program**, and these expressions are the program's **source code**. 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 that will print the result of **2 + 3** to the MathPiper console (**Note: the semicolon ";" that 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:

```
%mathpiper

2 + 3;

%/mathpiper
```

```
    %output,preserve="false"
      Result: 5
.   %/output
```

The most common type of output fold is **%output**, and by default folds of type %output have their **preserve property** set to **false**. This tells MathPiperIDE to overwrite the %output fold with a new version during the next execution of its parent. If preserve is set to **true**, the fold will not be overwritten, and a new fold will be created instead.

There are other kinds of child folds, but in the rest of this document they will all be referred to in general as "output" folds.

8.1.1 The title Attribute

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

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

2 + 3;

%/mathpiper
```

The title 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 **title="<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 top and bottom fold lines (for example:

```
%mathpiper

%/mathpiper
```

can be tedious so MathPiperIDE has a way to automatically insert them. Place the cursor at the beginning of a blank line in a .mpws 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 that 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 .mpws 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 that have a "**preserve="false"**" property will be removed.

8.3 Placing Text Outside Of A Fold

Text can also be placed outside of a fold like the following example shows:

```
Text can be placed above folds like this.

text text text text
text text text text

%mathpiper,title="Fold 1"

2 + 3;

%/mathpiper

Text can be placed between folds like this.

text text text text
text text text text

%mathpiper,title="Fold 2"

3 + 4;

%/mathpiper

Text can be placed after folds like this.

text text text text
text text text text
```

Placing text above a fold is useful for describing what is being done inside the fold.

8.4 Rectangular Selection Mode And Text Area Splitting

8.4.1 Rectangular Selection Mode

One capability that MathPiperIDE 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 three or four lines of text into a text area.

2) Hold down the **<Alt>** key (or the **<control>** key on Macintosh computers) then slowly press the **backslash key** (****) a few times. The bottom of the MathPiperIDE window contains a text field that MathPiperIDE uses to communicate information to the user. As **<Alt>** is repeatedly pressed, messages are displayed that 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.**

4) Holding down the **<CTRL>** key (or the **<command>** key on Macintosh computers) in regular selection mode will temporarily place the system into rectangular selection mode.

Most of the time normal selection mode is what you want to use, but in certain situations rectangular selection mode is better.

8.4.2 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. MathPiperIDE has this ability and it is called **splitting**.

If you look just to the right of the toolbar there is an icon that looks like a blank window, an icon to the right of it that looks like a window that was split horizontally, and an icon to the right of the horizontal one that is split vertically. If you let your mouse hover over these icons, a short description will be displayed for each of them.

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.

8.4.3 Exercises

A MathPiperIDE worksheet file called "**intro_book_examples_1.mpws**" is included in the mathpiperide/examples directory and **it is opened by default** when the software is first launched after it is downloaded. It contains a number of %mathpiper folds that 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 MathPiperIDE.

Then, use the worksheet to do the following exercises.

8.4.3.1 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.

8.4.3.2 Exercise 2

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

8.4.3.3 Exercise 3

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

8.4.3.4 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 that was calculated in d).

9 MathPiper Programming Fundamentals

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

9.1 Values, Literals, And Expressions

A **value** is a single symbol or a group of symbols that 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!"

is a "string" of characters that represents an English sentence (strings are covered in a later section).

A **literal** is any notation in computer source code that represents a value. Any number that is present in the source code of a program is a literal. For example, the 3 above is an integer number literal, and the number 0.5 is a real number literal. Additional literals will be discussed in later sections.

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

3

2 + 3

5 + 6*21/7 - 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 **3** to the integer **2** and return the result.

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

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

```
In> 5 - 2
Result: 3

In> 3*4
Result: 12

In> 30/3
Result: 10

In> 11%5
Result: 1

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) that are implemented with **procedures**. One can either call the procedure that an operator represents directly, or use the operator to call the procedure 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 (right associative).
/	Then division (left associative).
*	Then multiplication (left associative).
%	Then the remainder operator (left associative).
+, −	Finally, addition and subtraction (left associative).

This multi-operator expression will be used as an example to illustrate the precedence rules.

1) source code form:

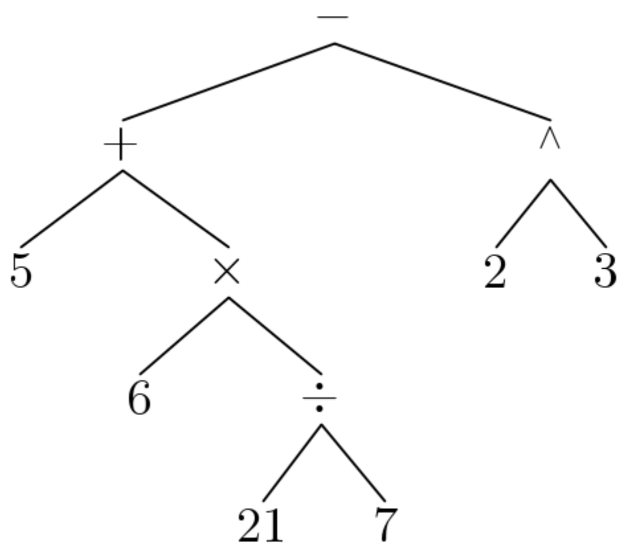
```
5 + 6*21/7 - 2^3
```

2) traditional mathematics form:

$$5 + 6 \times \frac{21}{7} - 2^3$$

3) expression tree form:

```
In> Show(TreeView('(5 + 6*21/7 - 2^3)))
Result: class javax.swing.JFrame
```



The **'** operator in the above code is named the “hold” operator, and it prevents an expression from being evaluated. In the following code, the **'** operator is used to prevent the expression 2 + 3 and the variable 'a' from being evaluated:

```
In> '(2 + 3)
Result: 2+3

In> a := 3
Result: 3

In> 'a
Result: a
```

The “hold” operator is useful when one wants to work with an expression instead of the value that the expression returns.

MathPiper uses **post-order** evaluation of expressions instead of **PEMDAS**. This is how post-order evaluation works (See http://patternmatics.com/test/expression_structure.html):

1) Start with the operator that is at the top of the expression tree.

2) Evaluate the operator's left subtree.

3) Evaluate the operator's right subtree.

4) Evaluate the operator.

Let's manually apply the precedence rules and post-order evaluation to the multi-operator expression we used earlier.

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

```
5 + 6*21/7 - 2^3
5 + 6*3 - 2^3
5 + 18 - 2^3
23 - 2^3
23 - 8
4
```

Starting with the first line, MathPiper evaluates the **/** operator first, which results in the **3** in the line below it. In the second line, the ***** operator is executed next, and so on. The last line shows that the final result after all of the operators have been evaluated is **15**.

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 2 + 4, 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 procedures.)

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.
^	Exponents (right associative).
/	Then division (left associative).
*	Then multiplication (left associative).
%	Then the remainder operator (left associative).
+, −	Finally, addition and subtraction (left associative).

9.5 Procedures & Procedure Names

In programming, **procedures** are named sequences 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. Procedures **can have values passed to them** from the calling code (called **arguments**), and they **always return a value** back to the calling code when they are finished executing. An example of a procedure is the **Even?()** procedure, which was discussed in an previous section.

Procedures 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 sequences of code are sometimes called **subroutines**, **procedures**, or **methods**.

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

9.6 Procedures That Produce Side Effects

Most procedures are executed to obtain the **results** they produce, but some procedures are executed in order to **have them perform work that is not in the form of a result**. Procedures 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 procedure produces a side effect that 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()** procedures are examples of procedures that produce side effects, and they are covered in the next section.

9.6.1 Printing Related Procedures: Echo(), Write(), And Newline()

The printing related procedures send text information to the user and this is usually referred to as "printing" in this document. However, it may also be called "echoing" and "writing".

9.6.1.1 The Echo() Procedure

The **Echo()** procedure 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() procedure, 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 procedures return a result, but procedures 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.

Each time an Echo() procedure 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() procedure 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, let's 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 assigned to all three variables, only the last variable's value was displayed.

The following example shows how Echo() can be used to display the values that are assigned to 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 Echo Procedures Are Useful For "Debugging" Programs

The errors that are in a program are often called "bugs". This name came from the days when computers were the size of large rooms and were made using electromechanical parts. Periodically, bugs would crawl into the machines and interfere with its moving mechanical parts and this would cause the machine to malfunction. The bugs needed to be located and removed before the machine would run properly again.

Of course, even back then most program errors were produced by programmers entering wrong programs or entering programs wrong, but they liked to say that all of the errors were caused by bugs and not by themselves! The process of fixing errors in a program became known as **debugging** and the names "bugs" and "debugging" are still used by programmers today.

One of the standard ways to locate bugs in a program is to place **Echo()** procedure calls in the code at strategic places that **print the contents of variables and display messages**. These Echo() procedures will enable you to see what your program is doing while it is running. After you have found and fixed the bugs in your program, you can remove the debugging Echo() procedure calls or comment them out if you think they may be needed later (comments are covered in a later section).

9.6.1.3 Write()

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

```
%mathpiper

Write(1);

Write(2);

Echo(3);
```

```
%/mathpiper

    %output,preserve="false"
      Result: True

      Side Effects:
      123
.   %/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 procedures.

9.6.1.4 NewLine()

The **NewLine()** procedure starts a new line in the side effects output. It can be used to print blank lines, which are 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 evaluated 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 that 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:

```
%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
```

The spaces that are in the code 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:

```
%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
```

9.7.2 Placing Consecutive Expressions Into A Code Sequence

It is often useful to place a sequence of expressions that are used together to accomplish a task into a group. In MathPiper these groups are called "code sequences." A **code sequence** (which is also called a **compound expression**) consists of one or more expressions that are separated by semicolons and placed within an open brace (**{**) and close brace (**}**) pair. When a code sequence is evaluated, each expression in the sequence will be executed from left-to-right or top-to-bottom. The following example shows expressions being executed within a code sequence:

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

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

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

Finally, code sequences can have their contents placed on separate lines if desired:

```
%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
```

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

9.7.2.1 Automatic Bracket, Parentheses, And Brace Match Indicating

In programming, most open brackets '**[**' have a close bracket '**]**', most open parentheses '**(**' have a close parentheses '**)**', and most open braces '**{**' have a close brace '**}**'. It is often difficult to make sure that each "open" character has a matching "close" character and if any of these characters don't have a match, then an error will be produced.

Thankfully, most programming text editors have a character match indicating tool that will help locate problems. To try this tool, paste the following code into a .mpws file and follow the directions that are present in its comments:

```
%mathpiper

/*
    Copy this code into a .mpws file.  Then, place the cursor
    to the immediate right of any {, }, [, ], (, or ) character.
    You should notice that the match to this character is
    indicated by a rectangle being drawing around it.
*/


list := [1,2,3];

{
    Echo("Hello");

    Echo(list);
}

%/mathpiper
```

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 that is enclosed within double quotes**. Text in a program's source code that is enclosed in double quotes is called a **string literal**. Strings can be assigned to variables just like numbers can, and strings can also be displayed using the Echo() procedure. The following program assigns a string value to the variable 'a' and then prints 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 MathPiperIDE 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 assigned 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() procedure 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() procedure 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
```

9.8.2.1 Combining Strings With The + Operator

If you need to combine two or more strings into one string, you can use the **+** operator like this:

```
In> "A" + "B" + "C"
Result: "ABC"

In> "Hello " + "there!"
Result: "Hello there!"
```

9.8.2.2 WriteString()

The **WriteString()** procedure prints a string without showing the double quotes that are around it. For example, here is the Write() procedure being used to print the string "Hello":

```
In> Write("Hello")
Result: True
Side Effects:
"Hello"
```

Notice the double quotes? Here is how the WriteString() procedure prints "Hello":

```
In> WriteString("Hello")
Result: True
```

```
Side Effects:
Hello
```

9.8.2.3 Nl()

The **Nl()** (New Line) procedure is used with the + procedure to place newline characters inside of strings:

```
In> WriteString("A" + Nl() + "B")
Result: True
Side Effects:
A
B
```

9.8.2.4 Space()

The Space() procedure is used to add spaces to printed output:

```
In> WriteString("A"); Space(5); WriteString("B")
Result: True
Side Effects:
A     B

In> WriteString("A"); Space(10); WriteString("B")
Result: True
Side Effects:
A          B

In> WriteString("A"); Space(20); WriteString("B")
Result: True
Side Effects:
A                    B
```

9.8.3 Accessing The Individual Letters/Characters 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 assigned 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"
```

9.8.3.1 Indexing Before The Beginning Of A String Or Past The End Of A String

Let's see what happens if an index is used that is less than **1** or greater than the length of a given string. First, we will assign the string "Hello" to the variable 'a':

```
In> a := "Hello"
Result: "Hello"
```

Then, we'll index the character at position **1** and then the character at position **0**:

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

In> a[0]
Result:
Exception: In procedure "StringMidGet" :
bad argument number 1(counting from 1) :

The offending argument aindex evaluated to 0
 In procedure: Nth,
```

Notice that using an index of **0** resulted in an error.

Next, let's access the character at position **5** (which is the 'o'), and finally the character at position **6**:

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

In> a[6]
Result:
Exception: String index out of range: 8
```

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 that use slashes:

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

x := 2; //The variable x becomes 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 that 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 How To Tell If MathPiper Has Crashed And What To Do If It Has

Sometimes code will be evaluated that has one or more unusual errors in it, and the errors will cause MathPiper to "crash". Unfortunately, beginners are more likely to crash MathPiper than more experienced programmers are because a beginner's program is more likely to have errors in it. When MathPiper crashes, no harm is done but it will not work correctly after that. **The only way to recover from a MathPiper crash is to exit MathPiperIDE and then relaunch it.** All the information in your buffers will be saved and preserved **but the contents of the console will not be**. Be sure to copy the contents of the console into a buffer and then save it before restarting.

One way to tell if MathPiperIDE has crashed is that it will indicate that **there are errors in lines of code that are actually fine**. If you are receiving an error in code that looks okay to you, simply restarting MathPiperIDE may fix the problem. If you restart MathPiperIDE and the error is still present, this usually means that there really is an error in the code.

9.11 Exercises

For the following exercises, create a new MathPiperIDE worksheet file called **book_1_section_9_exercises_<your first name>_<your last name>.mpws**. (**Note: there are no spaces in this file name**). For example, John Smith's worksheet would be called:

book_1_section_9_exercises_john_smith.mpws.

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

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

//Sample fold.

%/mathpiper
```

If an exercise uses the MathPiper console instead of a fold, copy the work you did in the console into a text file so it can be saved.

9.11.1 Exercise 1

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

```
2 + 3 * 4
```

9.11.2 Exercise 2

Place the following calculations into a single MathPiper fold, and then use one Echo() procedure per variable to print the results of the calculations. Put strings in the Echo() procedures that indicate which variable each calculated value is assigned 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);
```

9.11.3 Exercise 3

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

```
a := (2*2*2*2*2);
b := (2^5);
c := (_x^2 * _x^3);
d := (2^2 * 2^3);
```

9.11.4 Exercise 4

The following code assigns a string that contains all of the upper case letters of the alphabet to the variable **upper**. Each of the three Echo() procedures 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() procedures 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]);
```

9.11.5 Exercise 5

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

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

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

9.11.6 Exercise 6

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

```
%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 Lists

The **list** value type is designed to hold expressions in an **ordered collection**. Lists are very flexible and they are one of the most heavily used value types in MathPiper. Lists can **hold expressions of any type**, they can **grow and shrink as needed**, and they can be **nested**. Expressions in a list can be **accessed by their position** in the list (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 brackets []**. When this notation is present in a program's source code, it is called a **list literal**. In the following example, a list is created that contains various expressions and then it is assigned to the variable **exampleList**:

```
In> exampleList := [7,42,"Hello",1/2,_var]
Result: [7,42,"Hello",1/2,_var]

In>  exampleList
Result: [7,42,"Hello",1/2,_var]
```

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

```
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 assigned 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> exampleList[1]
Result: 7

In> exampleList[2]
Result: 42

In> exampleList[3]
Result: "Hello"

In> exampleList[4]
Result: 1/2

In> exampleList[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 **unassigned variable**.

Lists can also hold other lists as shown in the following example:

```
In> exampleList := [20, 30, [31, 32, 33], 40]
Result: [20,30,[31,32,33],40]

In> exampleList[1]
Result: 20

In> exampleList[2]
Result: 30

In> exampleList[3]
Result: [31,32,33]

In> exampleList[4]
Result: 40

```

The expression in the **3rd** position in the list is another **list** that contains the integers **31**, **32**, and **33**.

An expression in this second list can be accessed by **two sets of brackets**:

```
In> exampleList[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.

10.1 Append!()

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

The **Append!()** procedure 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]
```

11 Random Integer Values

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 procedure for randomly obtaining nonnegative integers is discussed along with how to use it to simulate the rolling of dice.

11.1 Obtaining Random Integers With The RandomInteger() Procedure

One way that a MathPiper program can generate random integers is with the **RandomInteger()** procedure. The RandomInteger() procedure takes an integer as an argument and it returns a random integer between 1 and the passed in integer. The following example shows random integers between 1 and 5 **inclusive** being obtained from RandomInteger(). **Inclusive** here means that both 1 and 5 are included in the range of random integers that may be returned. If the word **exclusive** was used instead, this would mean that neither 1 nor 5 would be in the range.

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

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

```
In> RandomInteger(100)
Result: 15
In> RandomInteger(100)
Result: 14
```

```
In> RandomInteger(100)
Result: 82
In> RandomInteger(100)
Result: 93
In> RandomInteger(100)
Result: 32
```

A range of random integers that does not start with 1 can also be generated by using the **two argument** version of **RandomInteger()**. For example, random integers between 25 and 75 can be obtained by passing RandomInteger() the lowest integer in the range and the highest one:

```
In> RandomInteger(25, 75)
Result: 28
In> RandomInteger(25, 75)
Result: 37
In> RandomInteger(25, 75)
Result: 58
In> RandomInteger(25, 75)
Result: 50
In> RandomInteger(25, 75)
Result: 70
```

11.2 Simulating The Rolling Of Dice

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

```
In> RandomInteger(6)
Result: 5
In> RandomInteger(6)
Result: 6
In> RandomInteger(6)
Result: 3
In> RandomInteger(6)
Result: 2
In> RandomInteger(6)
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 sequence**. The following code outputs the sum of the two simulated dice:

```
In> {a := RandomInteger(6); b := RandomInteger(6); a + b;}
Result: 6
In> {a := RandomInteger(6); b := RandomInteger(6); a + b;}
Result: 12
In> {a := RandomInteger(6); b := RandomInteger(6); a + b;}
Result: 6
```

```
In> {a := RandomInteger(6); b := RandomInteger(6); a + b;}
Result: 4
In> {a := RandomInteger(6); b := RandomInteger(6); a + b;}
Result: 3
In> {a := RandomInteger(6); b := RandomInteger(6); 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.

11.3 Exercises

For the following exercises, create a new MathPiperIDE worksheet file called **book_1_section_11_exercises_<your first name>_<your last name>.mpws** (**Note: there are no spaces in this file name**). For example, John Smith's worksheet would be called:

book_1_section_11_exercises_john_smith.mpws.

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

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

//Sample fold.

%/mathpiper
```

If an exercise uses the MathPiper console instead of a fold, copy the work you did in the console into a text file so it can be saved.

11.3.1 Exercise 1

Create a program that will roll two simulated dice 20 times, and print the results of each of these rolls.

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 that have the ability to make decisions are able to solve a wider range of problems than programs that can't make decisions.

12.1 Relational Operators

A program's decision making ability is based on a set of special operators that are called **relational operators**. Another name for them is **comparison operators**, but we will call them relational operators in this book. A **relational operator** is an operator that is used to **compare two values**. Expressions that contain relational operators return a **boolean value** and a **boolean value** is one that can only be **True** or **False**. When the words "True" and "False" are present in a program's source code, they are called **boolean literals**. In case you are curious about the strange name, boolean values come from the area of mathematics called **boolean logic**. This logic was created by a mathematician named **George Boole** and this is where the name boolean came from. Table 2 shows the relational 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: Relational Operators

This example shows some of these relational 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 relational 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
```

Relational 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.

^ Exponents (right associative).

/ Then division (left associative).

* Then multiplication (left associative).

% Then the remainder operator (left associative).

+, − Addition and subtraction (left associative).

=?,!=?,<?,<=?,>?,>=? Finally, relational operators are evaluated (left associative).

12.2 Predicate Expressions

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

12.3 Exercises

For the following exercises, create a new MathPiperIDE worksheet file called **book_1_section_12a_exercises_<your first name>_<your last name>.mpws**. (**Note: there are no spaces in this file name**). For example, John Smith's worksheet would be called:

book_1_section_12a_exercises_john_smith.mpws.

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

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

//Sample fold.

%/mathpiper
```

If an exercise uses the MathPiper console instead of a fold, copy the work you did in the console into a text file so it can be saved.

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> NM(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() Procedure & Predicate Expressions

Most programming languages have the ability to make decisions, and the most commonly used procedure for making decisions in MathPiper is the **If()** procedure:

```
With code sequence body:

If(predicate)
{
    body_expressions
}

Without code sequence body:

If(predicate) body_expression;
```

Notice that in bodied procedures, the ; is placed after the closing }, not after the closing).

The expression or expressions that are contained in a If() procedure are called its "**body**", and all procedures that have bodies are called "**bodied**" procedures. If a body contains more than one expression, then these expressions need to be placed within a **code sequence** (code sequences were discussed in an earlier section). What a procedure's body is will become clearer after studying some example programs.

The way the If() procedure works is it evaluates the "**predicate**" expression that is passed to it as an argument, and then it looks at the value that the expression returns. If this value is **True**, the body of the If() procedure is executed. If the predicate expression evaluates to **False**, the body is not executed. (Note: any procedure that accepts a predicate expression as a parameter can also accept the boolean values True and False).

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

```
%mathpiper

number := 6;

If(number >? 5)
{
    Echo(number, "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, number has been set to 6 and therefore the expression number >? 5 is **True**. When the **If()** procedures evaluates the **predicate expression** and determines it is **True**, it then executes the **Echo()** procedure that is **in its body**. The **second Echo()** procedure at the bottom of the program prints "End of program" regardless of what the If() procedure does.

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

```
%mathpiper

number := 4;

If(number >? 5)
{
```

```
    Echo(number, "is greater than 5.");
}

Echo("End of program.");

%/mathpiper

    %output,preserve="false"
      Result: True

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

This time the expression **number >? 5** returns a value of **False**, which causes the **If()** procedure to not execute its body.

This version of the program contains an If() procedure that does not use a code sequence as a body:

```
%mathpiper

number := 4;

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

Echo("End of program.");

%/mathpiper

    %output,preserve="false"
      Result: True

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

If the := operator is used in the body of an If() procedure **that does not use a code sequence for its body**, the unbodied expression must be placed within parentheses:

```
%mathpiper

number := 6;

If(number >? 5) (number := 0);
```

```
number;

%/mathpiper

    %output,preserve="false"
      Result: 0

.   %/output
```

12.4.1 One If() Procedure Used With One Else Operator

An If() procedure can be used with an **Else** operator to evaluate one body if a predicate expression is True, and an alternative body if the predicate expression is False. The format for If/Else code is as follows:

```
If(predicate)
{
    evaluate_this_body_if_True.
}
Else
{
    evaluate_this_body_if_False
}
```

The following program prints “4 is NOT greater than 5” because the predicate x >? 5 is False:

```
%mathpiper

x := 4;

If(x >? 5)
{
    Echo(x,"is greater than 5.");
}
Else
{
    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 The &?, |?, And !? Boolean Operators

12.5.1 The &? "And" Operator

Sometimes a programmer needs to check if two expressions are **True** and one way to do this is with the **&?** operator (which is read "**and**"). This is the calling format for the &? operator:

```
expression1 &? expression2
```

If **both** of these expressions return a value of **True**, the **&?** operator will also return a **True**. However, if **either** of the expressions return a **False**, then the &? operator will return a **False**. This can be seen in the following example:

```
In> True &? True
Result: True

In> True &? False
Result: False

In> False &? True
Result: False

In> False &? False
Result: False

In> True &? True &? True &? True
Result: True
```

The following program demonstrates the &? operator being used:

```
%mathpiper

a := 7;
b := 9;

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

If(a >? 5 &? 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
```

12.5.2 The |? "Or" Operator

The **|?** operator (which is read "**or**") is similar to the &? operator in that it only works with predicate expressions. However, instead of requiring that both expressions be **True** in order to return a **True**, |? will return a **True** if **one or both expressions are True**.

Here is the calling format for |?:

```
expression1 |? expression2
```

This example shows the |? operator being used:

```
In> True |? True
Result: True

In> True |? False
Result: True

In> False |? True
Result: True

In> False |? False
Result: False

In> False |? False |? True |? False
Result: True
```

The following program also demonstrates the |? operator being used:

```
%mathpiper
```

```
a := 7;
b := 9;

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

If(a <? 5 |? 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
```

12.5.3 The !? "Not" Operator

The **!?** operator (which is read "not") works with predicate expressions like the &? and |? operators do, except it can only accept **one** expression as input. The way !? works is that it changes a **True** value to a **False** value and a **False** value to a **True** value. Here is the !? operator's calling format:

```
!? expression
```

These are examples of Not> being used:

```
In> !? True
Result: False

In> !? False
Result: True
```

The following is a program that uses the !? operator:

```
%mathpiper
```

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

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

%/mathpiper

    %output,preserve="false"
      Result: True

      Side Effects:
      3 =? 3 is True
      !? 3 =? 3 is False
.   %/output
```

12.6 Exercises

For the following exercises, create a new MathPiperIDE worksheet file called **book_1_section_12c_exercises_<your first name>_<your last name>.mpws**. (**Note: there are no spaces in this file name**). For example, John Smith's worksheet would be called:

book_1_section_12c_exercises_john_smith.mpws.

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

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

//Sample fold.

%/mathpiper
```

If an exercise uses the MathPiper console instead of a fold, copy the work you did in the console into a text file so it can be saved.

12.6.1 Exercise 1

Write a program that uses the RandomInteger() procedure to simulate the flipping of a coin (Hint: you can use 1 to represent a head and 2 to represent a tail.) Use predicate expressions, the If() procedure, and the Echo() procedure 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.

12.6.2 Exercise 2

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 that 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.
*/

die1 := RandomInteger(6);
die2 := RandomInteger(6);

Echo("Die1: ", die1, "  Die2: ", die2);
NewLine();

If( die1 <=? 3 &? die2 <=? 3)
{
    Echo("Both dice came up <=? to 3.");
}

%/mathpiper
```

12.6.3 Exercise 3

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 that 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.
*/

die1 := RandomInteger(6);
die2 := RandomInteger(6);

Echo("Die1: ", die1, "  Die2: ", die2);
NewLine();

If( die1 <=? 3 |? die2 <=? 3)
{
    Echo("At least one die came up <=? 3.");
}

%/mathpiper
```

13 The While() And Until() Looping Procedures

13.1 The While() Looping Procedure

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** procedure. The calling format for the **While** procedure is as follows:

```
While(predicate)
{
    body_expressions
}
```

Notice that in bodied procedures, the ; is placed after the closing }, not after the closing).

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

13.1.1 Printing The Integers From 1 to 10

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

```
%mathpiper

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


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

While(count <=? 10)
{
    Echo(count);

    count := (count + 1);  //Increment count 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 **count** is created. It is used to tell the Echo() procedure which integer to print, and it is also used in the predicate expression that determines if the While() procedure should continue to **loop** or not.

When the program is executed, **1** is assigned to **count,** and then the While() procedure is called. Notice that 1 is assigned to the variable **count above the While loop**. Assigning an initial value to a variable is called **initializing** the variable and in this case, count needs to be initialized before it is used in the While() procedure. The predicate expression **count <=? 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() procedure 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() procedure prints the current contents of count (which is 1) and then the expression count := (count + 1) is executed.

The expression **count := (count + 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 **count** is to say that it **increments count** by **1**.

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

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

13.1.2 Placing The Integers From 1 to 50 In A List

The previous program can be adjusted in a number of ways to achieve different results. For example, the following program places the integers from 1 to 50 into a list by changing the **10** in the predicate expression to **50** and changing the Write procedure to a **Append!**() procedure.

```
%mathpiper

// Place the integers from 1-50 in a list.

integersList := [];

count := 1;

While(count <=? 50)
{
    Append!(integersList, count);

    count := (count + 1);  //Increment count by 1.
}

integersList;

%/mathpiper

    %output
      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]
.   %/output
```

(Note: In MathPiperIDE, the above numbers will all be on a single line.)

13.1.3 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
```

13.1.4 Placing The Integers From 1 To 100 In Reverse Order Into A List

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

```
%mathpiper

// Place the integers from 1 to 100 in reverse order into a list.

resultList := [];

x := 100;

While(x >=? 1)
{
    Append!(resultList, x);

    x := (x - 1);  //Decrement x by 1.
}

resultList;

%/mathpiper

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

In order to achieve the reverse ordering, this program had to initialize (which means to assign an initial value to a variable) **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.

13.2 The Until() Looping Procedure

The While() procedure evaluates the predicate expression that is passed to it, and then it evaluates its body if the predicate is **True**, and it does not evaluate its body if its predicate is **False**. The Until() procedure is similar to the While() procedure, except it evaluates its body before it evaluates the predicate expression that is passed to it, and it continues looping until the predicate expression becomes **True** instead of **False**. Since Until() evaluates its body before it evaluates the predicate expression, its body is always evaluated at least once.

The calling format for the **Until** procedure is as follows:

```
Until(predicate)
{
    body_expressions
}
```

Notice that in bodied procedures, the ; is placed after the closing }, not after the closing).

13.2.1 Printing The Integers From 1 to 10

The following program uses a Until() procedure to print the integers from 1 to 10:

```
%mathpiper

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


/*
    Initialize the variable count to 1
    outside of the Until "loop".
*/
count := 1;

Until(count =? 11)
{
    Echo(count);

    count := (count + 1);  //Increment count by 1.
}

%/mathpiper
```

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

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

13.3 Expressions Inside Of Code Sequences Are Indented

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

13.4 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 "**Halt Evaluation**" button, which in the upper left corner of the console.

Let's experiment with the **Halt Evaluation** button by executing a program that contains an infinite loop and then stopping it:

```
%mathpiper

//Infinite loop example program.

x := 1;
While(x <? 10)
{
    x := 3; //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 **Halt Evaluation** button. When the program is interrupted, the %output fold will display the message "**User halted evaluation**" to indicate that the program was interrupted. After a program has been interrupted, the program can be edited and then rerun.

13.5 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 MathPiperIDE 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;
```

```
Echo("These are the rolls:");

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

/*
 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 procedure's predicate expression.
*/
While(roll <=? 50)
{
    //Roll the dice.
    die1 := RandomInteger(6);
    die2 := RandomInteger(6);


    //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() procedure so that it does not put too much text
     into the output fold.
    */
    Write(rollSum,',');


    /*
     These If() procedures determine which sum was rolled and then add
     1 to the variable that is keeping track of the number of times
     that sum was rolled. The bodies of these If() procedures are not in
     code sequences.
    */
    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
```

13.6 Exercises

For the following exercises, create a new MathPiperIDE worksheet file called **book_1_section_13_exercises_<your first name>_<your last name>.mpws**. (**Note: there are no spaces in this file name**). For example, John Smith's worksheet would be called:

book_1_section_13_exercises_john_smith.mpws.

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

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

//Sample fold.

%/mathpiper
```

If an exercise uses the MathPiper console instead of a fold, copy the work you did in the console into a text file so it can be saved.

13.6.1 Exercise 1

Create a program that uses a While loop to print the even integers from 2 to 50 inclusive.

13.6.2 Exercise 2

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

13.6.3 Exercise 3

Create a program that simulates the flipping of a **single 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 Procedures

A **predicate procedure** is a procedure that either returns **True** or **False**. Most predicate procedures in MathPiper have names that end with a question mark "**?**". For example, **Even?()**, **Odd?()**, **Integer?()**, etc. The following examples show some of the predicate procedures that are in MathPiper:

```
In> Even?(4)
Result: True

In> Even?(5)
Result: False

In> Zero?(0)
Result: True

In> Zero?(1)
Result: False

In> NegativeInteger?(-1)
Result: True

In> NegativeInteger?(1)
Result: False

In> Prime?(7)
Result: True

In> Prime?(100)
Result: False
```

There is also an **Assigned?()** predicate procedure that can be used to determine whether or not a value is assigned to a given variable:

```
In> State()
Result: []

In> Assigned?(a)
Result: False

In> a := 1
Result: 1

In> Assigned?(a)
Result: True

In> Unassign(a)
Result: True
```

```
In> State
Result: []

In> Assigned?(a)
Result: False
```

The complete list of predicate procedures is contained in the **Programming Procedures/Predicates** node in the MathPiperDocs plugin.

14.1 Finding Prime Numbers With A Loop

Predicate procedures 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 **Prime?()** procedure in order to determine which integers are prime and which integers are not prime:

```
%mathpiper

// Determine which integers between 1 and 20 (inclusive)
// are prime and which ones are not prime.

primes := [];

nonPrimes := [];

x := 1;

While(x <=? 20)
{
    primeStatus := Prime?(x);

    If(primeStatus =? True)
    {
        Append!(primes, x);
    }
    Else
    {
        Append!(nonPrimes, x);
    }

    x := (x + 1);
}

[primes, nonPrimes];

%/mathpiper

    %output
      Result: [[2,3,5,7,11,13,17,19],[1,4,6,8,9,10,12,14,15,16,18,20]]
```

```
.   %/output
```

This program worked fairly well, but it can be shortened by moving the **Prime?()** procedure **inside** of the **If()** procedure instead of using the **primeStatus** variable to communicate with it:

```
%mathpiper

// Determine which integers between 1 and 20 (inclusive)
// are prime and which ones are not prime.

primes := [];

notPrimes := [];

x := 1;

While(x <=? 20)
{
    If(Prime?(x) =? True)
    {
        Append!(primes, x);
    }
    Else
    {
        Append!(notPrimes, x);
    }

    x := (x + 1);
}

[primes, notPrimes];

%/mathpiper

    %output
      Result: [[2,3,5,7,11,13,17,19],[1,4,6,8,9,10,12,14,15,16,18,20]]
.   %/output
```

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

Strings can contain zero or more characters, and the **Length()** procedure 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()** procedure. The **Length()** procedure returned a **3**, which means the string contained **3 characters**.

The following example shows that strings can also be passed to procedures 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 ToString() Procedure

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 **ToString()** procedure so that its **leftmost** and **rightmost** digits can be assigned to **variables**:

```
In> number := 678
Result: 678

In> stringNumber := ToString(number)
Result: "678"

In> leftmostDigit := stringNumber[1]
Result: "6"

In> rightmostDigit := stringNumber[ Length(stringNumber) ]
Result: "8"
```

Notice that the Length() procedure is used here to determine which character in **stringNumber** held the **rightmost** digit. Also, keep in mind that when numbers are in string form, operations such as +, -, *, and / **cannot** be performed on them.

14.4 Finding Prime Numbers that End With 7 (And Multi-line Procedure Calls)

Now that we have covered how to turn a number into a string, let's use this ability inside a loop. The following program finds all the **prime numbers** between **1** and **500** that have a **7 as their rightmost digit**. Notice that it has one If() procedure placed inside of another If() procedure. Placing an If() procedure inside of another If() procedure is called **nesting**, and nesting is used to to make more complex decisions.

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

```
%mathpiper

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

resultList := [];

x := 1;

While(x <=? 500)
{
    If(Prime?(x))
    {
        stringVersionOfNumber := ToString(x);

        stringLength := Length(stringVersionOfNumber);

        //Notice that If() procedures can be placed inside of other
        // If() procedures.
        If(stringVersionOfNumber[stringLength] =? "7")
        {
            Append!(resultList, x);
        }

     }

    x := (x + 1);
}

resultList;

%/mathpiper

    %output
      Result: [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
```

14.5 Exercises

For the following exercises, create a new MathPiperIDE worksheet file called **book_1_section_14_exercises_<your first name>_<your last name>.mpws**. (**Note: there are no spaces in this file name**). For example, John Smith's worksheet would be called:

book_1_section_14_exercises_john_smith.mpws.

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

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

//Sample fold.

%/mathpiper
```

If an exercise uses the MathPiper console instead of a fold, copy the work you did in the console into a text file so it can be saved.

14.5.1 Exercise 1

Write a program that uses a While loop to determine **how many** prime numbers there are between 1 and 1000. **Do not print the numbers themselves, just how many there are**.

14.5.2 Exercise 2

Write a program that uses a While loop to print only the prime numbers between 10 and 99 that contain the digit 3 in **either** their ones place **or** their tens place.

15 More Applications Of Using While Loops With Lists

15.1 Adding 1 To Each Element In A List

Procedures 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 select each of the elements in an input list and return an output list that contains each of the elements in the input list increased by 1:

```
%mathpiper

// Add 1 to each element of a list.

list := [55,93,40,21,7,24,15,14,82];

listLength := Length(list);

index := 1;

While(index <=? listLength)
{
    list[index] := (list[index] + 1);

    index := (index + 1);
}

list;

%/mathpiper

    %output
      Result: [56,94,41,22,8,25,16,15,83]
.   %/output
```

15.2 Determining If A Number Is In A List

A **loop** can also be used to search through a list. The following program uses a **While()** and an **If()** to search through a list to see if it contains the number **53**. A message in a string is returned that indicates whether or not 53 was found in the list:

```
%mathpiper

//Determine if 53 is in the list.
```

```
testList := [18,26,32,42,53,43,54,6,97,41];

listLength := Length(testList);

result := "53 was not found in the list";

index := 1;

While(index <=? listLength)
{
    If(testList[index] =? 53)
    {
        result := "53 was found in the list at position " +
         ToString(index);
    }

    index := (index + 1);
}

result;

%/mathpiper

    %output
      Result: "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**.

15.3 Finding The Sum Of The Integers In A List Using A While Loop

```
%mathpiper

// Find the sum all all the integers in a list.

list := [5,10,8,1,6,4,7,7,15,2];

listLength := Length(list);

sum := 0;

index := 1;

While(index <=? listLength)
{
    sum := (sum + list[index]);

    index := index + 1;
}
```

```
sum;

%/mathpiper

    %output
      Result: 65
.   %/output
```

15.4 Exercises

For the following exercises, create a new MathPiperIDE worksheet file called **book_1_section_15a_exercises_<your first name>_<your last name>.mpws**. (**Note: there are no spaces in this file name**). For example, John Smith's worksheet would be called:

book_1_section_15a_exercises_john_smith.mpws.

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

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

//Sample fold.

%/mathpiper
```

If an exercise uses the MathPiper console instead of a fold, copy the work you did in the console into a text file so it can be saved.

15.4.1 Exercise 1

Create a program that uses a While loop and the Odd?() predicate procedure to analyze the following list and then print the **number** of odd numbers it contains. **Hint: think about using code similar to count := (count + 1) in order to do the counting.**

```
[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]
```

15.4.2 Exercise 2

Create a program that uses a While loop and a NegativeNumber?() procedure to **copy** all of the negative numbers in the following list into a **new list**. Use the variable **negativeNumbersList** to hold the new list. Print the contents of the list after it has been created.

```
[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]
```

15.4.3 Exercise 3

Create one program that uses a single While loop to analyze this list:

[73,12,80,37,56,94,40,21,7,24,15,14,82,93,32,74,22,68,65,52,85,61,46,86,25]

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 (**do not use the Sum() procedure**).

Hint: the following program finds the largest number in a list and it can be used as a starting point for solving this exercise.

```
%mathpiper

/*
 The variable that keeps track of the largest number encountered so
 far needs to be initialized to the lowest possible value it may
 hold.  Why?
*/

largest := 0;

numbersList := [4,6,2,9,7,1,3];

index := 1;

While(index <=? Length(numbersList) )
{
    Echo("Largest: ", largest);

    If(numbersList[index] >? largest)
    {
       largest := numbersList[index]);
    }

    index := (index + 1);
}

Echo("The largest number in the list is: ", largest);

%/mathpiper
```

15.5 The ForEach() Looping Procedure

The **ForEach()** procedure uses a **loop** to index through a list like the While() procedure does, but it is more flexible and automatic. ForEach() also uses bodied notation like the While() procedure 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 the body. Therefore, body is **executed once for each expression in the list**.

15.6 Print All The Values In A List Using A ForEach() procedure

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

%/mathpiper

    %output,preserve="false"
      Result: True

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

15.7 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()** procedure 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.
numbersSum := 0;

ForEach(number, [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.
    */
    numbersSum := (numbersSum + number);

    //Print the sum as it is being accumulated.
    Write(numbersSum,',');
}

NewLine(); NewLine();

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

%/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 is 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 that can be used to automatically place a large number of integers into a list with very little typing.

15.8 The .. Range Operator

```
first .. last
```

A programmer often needs to create a list that 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 .. 100
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.
```

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

The following program shows how to use a **ForEach()** procedure instead of a **While()** procedure 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() procedure instead of a While() procedure.  Notice that
  the ForEach() version requires less typing than the While()
  version.
*/

ForEach(number, 1 .. 100)
{
    If(Prime?(number)) Write(number,',');
}

%/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
```

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

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

```
%mathpiper

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

//Create a new list.
primesList := [];

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(Prime?(number))
    {
       primesList := Append(primesList, number);
    }
```

```
}

//Print information about the primes that were found.
WriteString("Primes: ");
Write(primesList);
NewLine();
Echo("The number of primes in the list is ", Length(primesList) );
Echo("The first number in the list is ", primesList[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 is 15
      The first number in the list is 2
.   %/output
```

As can be seen from the above examples, the **ForEach()** procedure 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 procedures that are even more powerful than these two.

15.9.2 Exercises

For the following exercises, create a new MathPiperIDE worksheet file called **book_1_section_15b_exercises_<your first name>_<your last name>.mpws**. (**Note: there are no spaces in this file name**). For example, John Smith's worksheet would be called:

book_1_section_15b_exercises_john_smith.mpws.

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

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

//Sample fold.

%/mathpiper
```

If an exercise uses the MathPiper console instead of a fold, copy the work you did in the console into a text file so it can be saved.

15.9.3 Exercise 1

Create a program that uses a **ForEach()** procedure and an **Odd?()** predicate procedure 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]

15.9.4 Exercise 2

Create a program that uses a **ForEach()** procedure and an **NegativeNumber?()** procedure to copy all of the negative numbers in the following list into a **new list**. Use the variable **negativeNumbersList** to hold the new list. Print the contents of the list after it has been created.

[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]

15.9.5 Exercise 3

Create one program that uses a single **ForEach()** procedure 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 (**do not use the Sum() procedure**).

[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]

15.9.6 Exercise 4

Create one program that does the following:

1) Use a **While loop** to make a list that contains **1000 random integers** between **1** and **100** inclusive.

2) Use a **ForEach()** loop to determine **how many** integers in the list you created are **prime** and use an **Echo()** procedure to print this total.

16 Procedures & Operators That Loop Internally

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

16.1 Procedures & Operators That Loop Internally To Process Lists

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

16.1.1 TableForm()

```
TableForm(list)
```

The **TableForm()** procedure 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?()** procedure 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 **!?** operator can also be used with predicate procedures like Contains?() to change their results to the opposite truth value:

```
In> !? Contains?([50,51,52,53,54,55,56,57,58,59], 75)
Result: True
```

16.1.3 Find()

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

The **Find()** procedure 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(list, predicate_procedure)
```

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

```
In> Select([46,87,59,-27,11,86,-21,-58,-86,-52], "PositiveInteger?")
Result: [46,87,59,11,86]
```

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

Here are some further examples that use the Select() procedure:

```
In> Select([16,14,82,92,33,74,99,67,65,52], "Odd?")
Result: [33,99,67,65]

In> Select([16,14,82,92,33,74,99,67,65,52], "Even?")
Result: [16,14,82,92,74,52]

In> Select(1 .. 75, "Prime?")
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() procedure to analyze.

16.1.6 The Nth() Procedure & The [] Operator

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

The **Nth()** procedure simply returns the expression that 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 Concat()

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

The Concat() procedure 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]
```

16.1.8 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]
```

16.1.9 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]
```

16.1.10 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 that 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]
```

16.1.11 FillList()

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

The FillList() procedure simply creates a list that 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]
```

16.1.12 RemoveDuplicates()

```
RemoveDuplicates(list)
```

RemoveDuplicates() removes any duplicate expressions that are contained 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]
```

16.1.13 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]
```

16.1.14 Partition()

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

The **Partition()** procedure 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: [[_a,_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.

16.1.15 BuildList()

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

The BuildList() procedure 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 that contains the integers 1 through 10:

```
In> BuildList(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> BuildList(x*2, x, 1, 10, 1)
Result: [2,4,6,8,10,12,14,16,18,20]
```

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

```
In> BuildList(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]
```

16.1.16 Sort()

```
Sort(list, compare)
```

Sort() sorts the elements of **list** into the order indicated by **compare** with compare typically being the **less than** operator "**<**" or the **greater than** operator "**>**":

```
In> Sort([4,7,23,53,-2,1], "<?");
Result: [-2,1,4,7,23,53]
```

```
In> Sort([4,7,23,53,-2,1], ">?");
Result: [53,23,7,4,1,-2]

In> Sort([1/2,3/5,7/8,5/16,3/32], "<?")
Result: [3/32,5/16,1/2,3/5,7/8]

In> Sort([.5,3/5,.76,5/16,3/32], "<?")
Result: [3/32,5/16,.5,3/5,.76]
```

16.2 Procedures That Work With Integers

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

16.2.1 RandomIntegerList()

```
RandomIntegerList(length, lowest_possible, highest_possible)
```

A vector is a list that does not contain other lists. **RandomIntegerList()** 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> RandomIntegerList(10, 1, 99)
Result: [73,93,80,37,55,93,40,21,7,24]
```

16.2.2 Maximum() & Minimum()

```
Maximum(value1, value2)
Maximum(list)
```

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

```
In> Maximum(10, 20)
Result: 20
```

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

```
In> testList := RandomIntegerList(10, 1, 99)
Result: [73,93,80,37,55,93,40,21,7,24]

In> Maximum(testList)
Result: 93
```

The **Minimum()** procedure is the opposite of the Maximum() procedure.

```
Minimum(value1, value2)
Minimum(list)
```

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

```
In> Minimum(10, 20)
Result: 10
```

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

```
In> testList := RandomIntegerList(10, 1, 99)
Result: [73,93,80,37,55,93,40,21,7,24]

In> Minimum(testList)
Result: 7
```

16.2.3 Quotient() & Modulo()

```
Quotient(dividend, divisor)
Modulo(dividend, divisor)
```

Quotient() determines the whole number of times a divisor goes into a dividend:

```
In> Quotient(7, 3)
Result: 2
```

Modulo() determines the **remainder** that results when a dividend is divided by a divisor:

```
In> Modulo(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()** procedure 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()** procedure 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 Sum()

```
Sum(list)
```

Sum() can find the sum of a list that is passed to it:

In> testList := RandomIntegerList(10,1,99)

```
Result: [73,93,80,37,55,93,40,21,7,24]
```

```
In> Sum(testList)
Result: 523
```

```
In> testList := (1 .. 10)
```

```
Result: [1,2,3,4,5,6,7,8,9,10]

In> Sum(testList)
Result: 55
```

16.2.7 Product()

```
Product(list)
```

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

```
In> Product([1,2,3])
Result: 6
```

16.3 Exercises

For the following exercises, create a new MathPiperIDE worksheet file called **book_1_section_16_exercises_<your first name>_<your last name>.mpws**. (**Note: there are no spaces in this file name**). For example, John Smith's worksheet would be called:

book_1_section_16_exercises_john_smith.mpws.

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

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

//Sample fold.

%/mathpiper
```

If an exercise uses the MathPiper console instead of a fold, copy the work you did in the console into a text file so it can be saved.

16.3.1 Exercise 1

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

Hint 1: You can use the following code as the starting point for your loop:

```
ForEach(num, 1 .. 5)
{

}
```

Hint 2: you can use the Sort() procedure to sort the generated list to make it easier to check if your program is counting correctly.

16.3.2 Exercise 2

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

16.3.5 Exercise 5

Create a program that uses **BuildList()** to obtain a list that 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 A Nested Loop

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.
*/

combinationsList := [];

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

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

TableForm(combinationsList);

%/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

For the following exercises, create a new MathPiperIDE worksheet file called **book_1_section_17_exercises_<your first name>_<your last name>.mpws**. (**Note: there are no spaces in this file name**). For example, John Smith's worksheet would be called:

book_1_section_17_exercises_john_smith.mpws.

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

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

//Sample fold.

%/mathpiper
```

If an exercise uses the MathPiper console instead of a fold, copy the work you did in the console into a text file so it can be saved.

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 Procedures

In computer programming, a **procedure** is a named section of code that can be **called** from other sections of code. **Values** can be sent to a procedure for processing as part of the **call**, and a procedure always returns a value as its result. A procedure 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 procedure when it is called are called **arguments** or **actual parameters**, and a procedure can accept 0 or more of them. These arguments are usually placed within parentheses.

MathPiper has many predefined procedures (some of which have been discussed in previous sections) but users can create their own procedures too. The following program creates a procedure called **addNums()** that 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 procedure called **addNums** and specified that it will accept two values when it is called. The **first** value will be assigned to the variable **num1** and the **second** value will be assigned to the variable **num2**.

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

The code on the **right side** of the **assignment operator** is **assigned** to the procedure name "**addNums**" and it is executed each time **addNums()** is called. The following example shows the new **addNums()** procedure 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 procedures that come with MathPiper, we chose to have this procedure'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 procedure names to begin with a lower case letter to distinguish them from the procedures that come with MathPiper**.

The values that are returned from user defined procedures can also be assigned to variables. The following example uses a %mathpiper fold to define a procedure called **evenIntegers()** and then this procedure is used in the MathPiper console to assign a list of even integers to the variable "a":

```
%mathpiper

evenIntegers(endInteger) :=
{
    resultList := [];

    x := 2;

    While(x <=? endInteger)
    {
        resultList := Append(resultList, x);

        x := (x + 2);
    }

    /*
     The result of the last expression that is executed in a procedure
     is the result that the procedure returns to the caller.  In this case,
     resultList is purposely being executed last so that its contents are
     returned to the caller.
    */
    resultList;
}

%/mathpiper

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

In> a := evenIntegers(10)
Result: [2,4,6,8,10]

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

The procedure **evenIntegers()** returns a list that 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()** procedure and make it ready for use. The **evenIntegers()** procedure was then called from the MathPiper console and 10 was passed to it.

After the procedure 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()** procedure in order to determine its size.

18.1 Global Variables, Local Variables, & Local()

The new **evenIntegers()** procedure seems to work well, but there is a problem. The variables '**x**' and **resultList** were defined inside the procedure as **global variables**, which means they are accessible from anywhere, including from within other procedures, 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 procedures is usually not a good idea because code in other procedures and folds might already be using (or will use) the same variable names. Global variables that 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 procedure named **Local()** can be called inside of a procedure to define what are called **local variables**. A **local variable** is only accessible inside the procedure 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()** procedure that 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);
    }

    /*
     The result of the last expression that is executed in a procedure
     is the result that the procedure returns to the caller.  In this case,
     resultList is purposely being executed last so that its contents are
     returned to the caller.
    */
    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> Unassign(x, resultList)
Result: True

In> evenIntegers(10)
Result: [2,4,6,8,10]

In> x
Result: x

In> resultList
```

```
Result: resultList
```

18.2 Exercises

For the following exercises, create a new MathPiperIDE worksheet file called **book_1_section_18_exercises_<your first name>_<your last name>.mpws**. (**Note: there are no spaces in this file name**). For example, John Smith's worksheet would be called:

book_1_section_18_exercises_john_smith.mpws.

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

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

//Sample fold.

%/mathpiper
```

If an exercise uses the MathPiper console instead of a fold, copy the work you did in the console into a text file so it can be saved.

18.2.1 Exercise 1

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

Hint: You may want to use the RandomIntegerList(), Select(), Odd?(), and Take() procedures.

18.2.2 Exercise 2

Create a procedure called **convertStringToList(string)** that takes a string as a parameter and returns a list that contains all of the characters in the string. Here is an example of how the procedure 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"]
```

Hint: Remember, a string can be broken down into individual characters by using an index value inside of brackets [] like this:

```
In> string := "Hello"
Result: "Hello"

In> string[1]
Result: "H"

In> string[2]
Result: "e"

In> string[3]
Result: "l"

In> string[4]
Result: "l"

In> string[5]
Result: "o"
```

Your procedure should use this indexing technique inside of a loop to append each of these characters to a list.

19 Miscellaneous topics

19.1 Incrementing And Decrementing Variables With The ++ And -- Operators

Up until this point we have been adding 1 to a variable with code in the form of **x := (x + 1)** and subtracting 1 from a variable with code in the form of **x := (x - 1)**. Another name for **adding** 1 to a variable is **incrementing** it and **decrementing** a variable means to **subtract** 1 from it. Now that you have had some experience with these longer forms, it is time to show you shorter versions of them.

19.1.1 Incrementing Variables With The ++ Operator

The number 1 can be added to a variable by simply placing the ++ operator after it like this:

```
In> x := 1
Result: 1

In> x++;
Result: 2

In> x
Result: 2
```

Here is a program that uses the **++** operator to increment a loop index variable:

```
%mathpiper

index := 1;

While(index <=? 10)
{
    Echo(index);

    index++; //The ++ operator increments the index variable.
}

%/mathpiper

    %output,preserve="false"
      Result: True

      Side Effects:
      1
```

```
      2
      3
      4
      5
      6
      7
      8
      9
      10
.   %/output
```

19.1.2 Decrementing Variables With The -- Operator

The number 1 can be subtracted from a variable by simply placing the **--** operator after it like this:

```
In> x := 1
Result: 1

In> x--;
Result: 0

In> x
Result: 0
```

Here is a program that uses the **--** operator to decrement a loop index variable:

```
%mathpiper

index := 10;

While(index >=? 1)
{
    Echo(index);

    index--; //The -- operator decrements the index variable.
}

%/mathpiper

    %output,preserve="false"
      Result: True

      Side Effects:
      10
      9
      8
      7
      6
```

```
        5
        4
        3
        2
        1
.   %/output
```

19.1.3 The For() Looping Procedure

The For() procedure provides an easy way to create loops that use an index variable. This is the calling format for the For() procedure:

```
For(initialization, predicate, changeIndex) body
```

The parameter named "initialization" is an expression that is usually used to assign an initial value to the index variable. The parameter named "predicate" is an expression that is evaluated before the body is evaluated. If this "predicate" evaluates to True, then the body is evaluated. If "predicate" evaluates to False, the body is not evaluated, and the For() procedure finishes. The parameter named "changeIndex" is used to increase or decrease the value that is assigned to the index variable.

The following code uses a For() procedure to print the integers from 1 to 10 inclusive:

```
%mathpiper

For(index := 1, index <=? 10, index++)
{
    Echo(index);
}

%/mathpiper

    %output,preserve="false"
      Result: True

      Side Effects:
      1
      2
      3
      4
      5
      6
      7
      8
      9
```

```
        10
.   %/output
```

19.1.4 The Break() Procedure

The **Break()** procedure is used to end a loop early and here is its calling format:

```
Break()
```

The following program has a While loop that is configured to loop 10 times. However, when the loop counter variable **index** reaches 5, the Break() procedure is called and this causes the loop to end early:

```
%mathpiper

index := 1;

While(index <=? 10)
{
    Echo(index);

    If(index =? 5) Break();

    index++;
}

%/mathpiper

    %output,preserve="false"
      Result: True

      Side Effects:
      1
      2
      3
      4
      5
.   %/output
```

When a Break() procedure is used to end a loop, it is called "**breaking out**" of the loop. Notice that only the numbers 1-5 are printed in this program.

19.1.5 The Continue() Procedure

The **Continue()** procedure is similar to the Break() procedure, except that instead of ending the loop, it simply causes it to **skip the remainder of the loop for the current loop iteration**. Here is the Continue() procedure's calling format:

```
Continue()
```

The following program uses a While loop that is configured to print the integers from 0 to 8. However, the Continue() procedure is used to skip the execution of the Echo() procedure when the loop indexing variable **index** is equal to 5:

```
%mathpiper

index := 0;

While(index <? 8)
{
    index++;

    If(index =? 5) Continue();

    Echo(index);
}

%/mathpiper

    %output,preserve="false"
      Result: True

      Side Effects:
      1
      2
      3
      4
      6
      7
      8
.   %/output
```

Notice that the number 5 is not printed when this program is executed.

19.1.6 The Repeat() Looping Procedure

The **Repeat()** procedure is a looping procedure that is similar to While() and ForEach(), but it is simpler than these two. Here are the two calling formats for Repeat():

```
Repeat(count) body
Repeat() body
```

The first version of Repeat() simply takes an integer argument that indicates how many times it should loop. The following program shows how to use Repeat() to print 4 copies of the word "Hello":

```
%mathpiper

Repeat(4)
{
    Echo("Hello");
}

%/mathpiper
```

```
    %output,preserve="false"
      Result: 4

      Side Effects:
      Hello
      Hello
      Hello
      Hello
.   %/output
```

The second version of Repeat() does not take any arguments and it is designed to run as an **infinite loop**. The Break() procedure is then used to make the Repeat() procedure stop looping. The following program would print the loop indexing variable **index** forever, but the Break() procedure is used to stop the loop after **3** iterations:

```
%mathpiper

index := 1;

loopCount := Repeat()
{
    Echo(index);

    If(index =? 3) Break();

    index := (index + 1);
}

```

```
Echo("Loop count: ", loopCount);

%/mathpiper

    %output,preserve="false"
      Result: True

      Side Effects:
      1
      2
      3
      Loop count: 2
.   %/output
```

Notice that Repeat() returns the number of times it actually looped as a result and that this value is assigned to the variable **loopCount**.

19.1.7 The EchoTime() Procedure

Computers are extremely fast, but they still take time to execute programs. Sometimes it is important to determine how long it takes to evaluate a given expression in order to do things like determine if a section of code need to run quicker than it currently is or determine if one piece of code is slower than another. The EchoTime() procedure is a bodied procedure that is used to **time how long a section of code takes to run**. Here is its calling format:

```
EchoTime()expression
```

The following examples use EchoTime() to determine how long it takes to add the numbers 2 and 3 together and how long it takes to factor 1234567:

```
In> EchoTime() 2 + 3
Result: 5
Side Effects:
0.000080946 seconds taken.


In> EchoTime() Factor(1234567)
Result: 127*9721
Side Effects:
0.395028773 seconds taken.
```

In the following program, a ForEach loop is used to have the Factor() procedure factor all the numbers in a list. The EchoTime() procedure is used to determine how long it takes to do all the factoring:

```
%mathpiper

EchoTime() ForEach(number, [100,54,65,67,344,98,454])
{
    Echo(number, " - ", Factor(number));
}

%/mathpiper

    %output,preserve="false"
      Result: True

      Side Effects:
      100  - 2^2*5^2
      54  - 2*3^3
      65  - 5*13
      67  - 67
      344  - 2^3*43
      98  - 2*7^2
      454  - 2*227
      0.262678978 seconds taken.
.   %/output
```

Finally, the following program shows how to time a code sequence that prints the numbers from 1 to 100:

```
%mathpiper

EchoTime()
{
    index := 1;

    While(index <=? 100)
    {
        Write(index,',');

        If(index % 10 =? 0) NewLine();

        index++;
    }

    NewLine();
}

%/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,

      0.055418423 seconds taken.
.   %/output
```

19.2 Exercises

For the following exercises, create a new MathPiperIDE worksheet file called **book_1_section_19_exercises_<your first name>_<your last name>.mpws**. (**Note: there are no spaces in this file name**). For example, John Smith's worksheet would be called:

book_1_section_19_exercises_john_smith.mpws.

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

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

//Sample fold.

%/mathpiper
```

If an exercise uses the MathPiper console instead of a fold, copy the work you did in the console into a text file so it can be saved.

19.2.1 Exercise 1

Create a program that uses a While loop to display the numbers from 1 to 50. Use the ++ operator to increment the loop index variable.

19.2.2 Exercise 2

Create a program that uses a While loop to display the numbers from 1 to 50 in reverse order. Use the -- operator to decrement the loop index variable.

19.2.3 Exercise 3

Create a program that uses a Continue() procedure to cause a While loop that is configured to print the numbers from 1 to 100 to skip printing the number 72.

19.2.4 Exercise 4

Create a program that uses the version of the Repeat() procedure that takes an integer as an argument and the + string concatenation operator to print the following:

```
Hello
HelloHello
HelloHelloHello
HelloHelloHelloHello
HelloHelloHelloHelloHello
```

Hint:

```
In> string := "Hi"
Result: "Hi"

In> string := string + "Hi"
Result: "HiHi"
```

19.2.5 Exercise 5

In the last example in the EchoTime() section, what operator is being used to format the output into lines of 10 numbers and how is this operator doing this?