LogoMation 2.0:

Prelude into Programming

and a

Reference Manual

Chuck Shavit
Magic Square, Inc.

June 1997
revised April 1998

Email inquiries: **info@MagicSquare.com**

Before We Begin

LogoMation is for everyone: kids can do wonders with it. Grownups can do impressive things with it, too. You will enjoy LogoMation if it is your first programming language, or if it is your tenth.

Yuval Shavit's dad developed LogoMation when Yuval was eight and a half. One day Yuval came up to his dad and asked to program a computer, just like mom and dad do. When Yuval's dad started developing LogoMation, Yuval could do many more things than LogoMation would allow him to do. LogoMation had since become smarter, and so did Yuval.

This book is divided into two parts: a tutorial for the beginner programmer, and a reference manual. The tutorial has 7 lessons, labeled A to G. The first lesson starts on page 4. The tutorial is for everyone: it is for kids who never wrote a computer program before, and for grownups who did not even *think* that they can write programs. It will introduce you to the fundamentals of computer programming, as well as the fundamentals of LogoMation.

People also enjoy LogoMation if they have programmed computers before. See, anyone can use LogoMation, even seasoned programmers. Experienced programmers might decide to quickly skim through the tutorial, and go directly to the reference manual. The first reference manual chapter, *A Quick Tour*, starts on page 38.

So what is LogoMation?

There are hundreds, perhaps thousands, of programming languages out there. They all seem to have one goal in common: to tell a computer how to execute certain operations. Typically, a given task can be programmed in many of these languages. The choice of which language to chose for a given task has to do with the ease of applying a particular programming language for the task at hand, with the effectiveness of the language for the task, and also with the programmer's personal preference.

LogoMation is a programming environment. Using LogoMation, you will discover that programming is fun, and is not as difficult as you may have thought it was.

With LogoMation, you write programs that tell the computer to draw things. When the program is ready, you click the **Go** button. Then bingo – you can see the picture that you were describing to the computer. The picture can be static (not moving), or animated (moving). It can even have sound effects. In short: your very own LogoMation creation.

Sounds too simple? Well, it's like skiing or playing the piano: if you love it, it is a little hard at first, but a lot of fun even as you start learning it. And as you learn more and more, it is even more fun.

Learning LogoMation

Learning a new programming language is not all that difficult – if you are already familiar with another language. Learning LogoMation is not difficult *even* if it is your first programming language. It was designed this way.

You may ask at this point: will I need a lot of math skills to use LogoMation? The answer is yes and no. No – you will not need to be a math wiz to draw real lovely pictures, which will give you a tremendous amount of pleasure and fun. And yes – you *will* need some math to draw some more complicated things. Whatever math skills you have, you can use LogoMation and have fun.

The tutorial will teach you some of the basics of programming in general, and LogoMation programming in particular. It is structured so that you can learn these basics all by yourself.

However, if LogoMation is your first programming language, it might help if you can occasionally get help from someone who is more experienced in programming.

Use the tutorial as your introduction to LogoMation. Follow the step-by-step instruction, and don't do it all at once. I am not telling you how many lessons you can take at one time, but I advise you not to take too many, or else it will confuse you too much. If you never wrote computer programs before, do not take more than one lesson a day. I also suggest that you *really* do the exercises, even though I am sure that you hate homework just like everybody else. These exercises are fun to do, and it is through these exercises that you'll learn LogoMation. The only way to really learn programming is by actually writing programs.

By the way, this book assumes that you know how to use your computer, e.g., double-click icons, select menu commands, and type and edit text.

One last thing before we start: you may have heard of the Logo language. Maybe you have actually programmed in that language. LogoMation borrowed from the original Logo language the idea of "turtle geometry": the idea that you are moving a pen around and thus draw on the screen. But other than borrowing the idea, LogoMation is totally different from Logo. So if you are familiar with Logo, I suggest that you still spend the time to learn LogoMation as if you have never used Logo.

LogoMation is available on the following configurations:

- PCs running Windows 95 and higher.
- PCs running Windows NT 4.0 and higher.
- Macintosh computers – both 68K-based and PowerPC-based – running O/S 6.0.7 or higher.

The Windows and Macintosh implementations of LogoMation are very similar. Where applicable, differences are highlighted in a box like this one.

Most of the screen shots in this manual were taken on a Window implementation. The Mac implementation is quite similar. The specific implementation on Microsoft Windows is described in *Chapter 11: Running LogoMation on Windows*, on page 116. The specific implementation on Macintosh computers is described in *Chapter 12: Running LogoMation on Macintosh*, on page 124.

Table of Contents

BEFORE WE BEGIN 2

LESSON A: LOGOMATION IS ABOUT MOVING 9

LESSON B: LET'S MOVE SOME MORE 13

LESSON C: TEXT, COLOR, AND FILL 19

LESSON D: CIRCLES 24

LESSON E: VARIABLES AND FUNCTIONS, PART I 26

LESSON F: VARIABLES AND FUNCTIONS, PART II 30

LESSON G: FUN WITH ANIMATION 34

CONGRATULATIONS! 37

CHAPTER 1: A QUICK TOUR 38

1.1 THE LOGOMATION WINDOWS 38

1.2 HELLO WORLD – THE FIRST LOGOMATION PROGRAM 38

1.3 WYSIWYG INDENTATION 40

1.4 THE *TRACER* 41

1.5 PROGRAM DEVELOPMENT AND DEBUGGING 41

1.6 ANIMATION AND SOUND 43

1.7 WHAT'S NEXT 43

CHAPTER 2: SYNTAX 44

2.1 A LOGOMATION PROGRAM 44

2.2 COMPOUND STATEMENTS 44

2.3 COMMENTS 45

CHAPTER 3: DATA TYPES, VARIABLES AND EXPRESSIONS 46

3.1 FOUR DATA TYPES 46

3.2 NUMBERS 46

3.3 STRINGS 46

3.4 VARIABLES 47

3.5 LISTS 48

3.6 ASSOCIATIVE ARRAYS 49

3.7 COMBINING INDEXING; MULTIDIMENSIONAL ARRAYS 50

3.8 THE TYPE OF DATA STORED IN VARIABLES 50

3.9 EXPRESSIONS 50

CHAPTER 4: THE GRAPHICS ENVIRONMENT 53

4.1 THE EDIT AND RUN WINDOWS 53

4.2 POINTS AND PENS 53

4.3 MOVEMENT MODES 53

4.4 DRAWING TO THE SCREEN *VS.* PRINTING 54

CHAPTER 5: STATEMENTS 55

5.1 RELATIVE PEN MOVEMENT STATEMENTS 55
5.1.1 The FORWARD Statement 55
5.1.2 The BACKWARD Statement 56
5.1.3 The RIGHT Statement 56
5.1.4 The LEFT Statement 56
5.1.5 The STRAIGHT Statement 57
5.1.6 The CIRCLE Statement 57

5.2 ABSOLUTE MOVEMENT, `UP` AND `DOWN` STATEMENTS 58
5.2.1 The GOTO Statement 58
5.2.2 The DOWN Statement 58
5.2.3 The UP Statement 59

5.3 PEN AND BACKGROUND PROPERTIES STATEMENTS 59
5.3.1 The COLOR Statement 59
5.3.2 The WIDTH Statement 60
5.3.3 The CLEAR Statement 60
5.3.4 The PATTERN Statement 60

5.4 PRINTING 61

5.5 HALTING THE PROGRAM'S EXECUTION 62

5.6 TEMPORARILY HALTING THE PROGRAM'S EXECUTION 63

5.7 PAUSING A PROGRAM'S EXECUTION 63

5.8 PLAYING A SOUND 63

5.9 LIBRARY FILES 64

5.10 LOGOMATION LOOPS 64
5.10.1 The REPEAT Statement 64
5.10.2 the BREAK Statement 65
5.10.3 The WHILE Statement 66

5.11 LOGOMATION CONDITIONALS – THE `IF…` AND `IF…ELSE…` STATEMENTS 66

5.12 THE `CURVE` STATEMENT 68

5.13 THE `FILL` STATEMENT 69

CHAPTER 6: FUNCTIONS 71

6.1 DEFINING A FUNCTION 71

6.2 CALLING A FUNCTION 71

6.3 RETURNING A VALUE 72

6.4 ARGUMENT BINDING 73

6.5 SCOPING 74

6.6 LOCAL VARIABLES 74

6.7 BUILT-IN FUNCTIONS 76
abs 76
acos 76
ascii 76
asin 77
ask 77
atan 77
atan2 78

bg_rgb 78
char 78
closeFile 79
cos 79
defined 79
dir 80
dirTo 80
distanceTo 80
exec 81
exp 81
form 81
format 82
free_stack 82
get_char 83
getFileName 83
hdc 84
height 84
hi 84
hwnd 84
indices 84
int 85
join 85
listlen 85
length 85
log 85
max 86
mci 86
mem 86
min 86
mouse 86
openFile 87
penAbove 87
penDuration 87
penEnd 87
penFontBold() or penBold() 88
penFontdir() 88
penFontItalic()or penItalic() 88
penFontname 88
penFontsize 88
penFontstyle() 89
penFontUnderline() or penUnderline() 89
penName 89
penPicture 89
penRadius 89
penSpeed 90
penTrail 90
penUp 90
penWidth 90
pop 90
push 90
random 91
randomize 91
readFile 92
rgb 92
round 92
sAscent 93
sDescent 93
seconds 93
shift 93

sin 94
split 94
sprintf 94
sqrt 95
sublist 95
substr 95
sWidth 95
tan 96
this 96
this0 96
this1 97
time 97
type 97
width 97
writeFile & writeFileNow 98
x 98
y 99

CHAPTER 7: READING AND WRITING FILES 100

7.1 SUMMARY **100**

7.2 TECHNIQUES & HINTS **100**
7.2.1 Typical File Reading Process 100
7.2.2 Typical File Writing Process 102
7.2.3 Closing Files 102
7.2.4 Hard-Coded Pathnames 102

CHAPTER 8: PENS 103

8.1 A PEN IS A SET OF ATTRIBUTES **103**

8.2 DEFINING AND SWITCHING PENS **103**

8.3 THE PEN'S ATTRIBUTES **104**
8.3.1 Font Attributes 104
8.3.2 Animation Attributes 106
8.3.3 Other Attributes 107

8.4 SETTING A PEN AS A *TRACER* **107**

CHAPTER 9: ANIMATION 109

9.1 THE `PICTURE` STATEMENT **109**

9.2 THE ANCHOR POINT **110**

9.3 MOVEMENT MODES **110**

9.4 SINGLE FRAME VS. MULTI FRAME PICTURES **111**

9.5 MULTI FRAME PICTURES **111**

CHAPTER 10: THE _LM_STARTUP FILE 114

10.1 THE STARTUP FILE **114**

10.2 `__LM_STARTUP()` **114**

10.3 APPLICATION – "TURTLE MODE" **114**

10.4 ANOTHER APPLICATION – "STATIONARY" BACKGROUND **115**

CHAPTER 11: RUNNING LOGOMATION ON WINDOWS 116

11.1 STARTING LOGOMATION, MENUS **116**

11.2 EDIT & RUN WINDOWS **117**

11.3 EDITING A PROGRAM 117
11.4 SOUNDS AND PICTURES IN LOGOMATION 121
11.5 RUNNING AND DEBUGGING A PROGRAM 122
11.6 PRINTING 123

CHAPTER 12: RUNNING LOGOMATION ON MACINTOSH 124
12.1 STARTING LOGOMATION, MENUS 124
12.2 EDIT & RUN WINDOWS 124
12.3 EDITING A PROGRAM 124
12.4 SOUNDS AND PICTURES IN LOGOMATION 127
12.5 RUNNING AND DEBUGGING A LOGOMATION PROGRAM 127
12.6 PRINTING 128
12.7 TRACER MODE 128
12.8 PREFERENCES 130
12.9 GETTING HELP 131

APPENDIX A. INSTALLATION.....132

APPENDIX B. FORMAT SPECIFICATIONS IN FORMAT() 134

Lesson A: LogoMation is About Moving

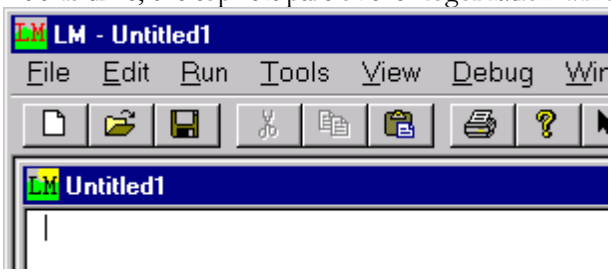
LogoMation moves things. When your program starts, there is a pen waiting to be moved on the screen and draw stuff for you. You can move the pen straight ahead, in a straight line, then you can tell the pen to turn left and move in a different direction. It is almost like the pen is a little creature that you can tell things to. You tell the pen to go forward 100, and it does just that: move 100 *pixels* on the screens. A *pixel* is a little dot on the screen. You can think of the screen as being made of lots and lots of dots, and each can be in a different color. So, when you tell the pen to move forward 100 pixels, it does just that, coloring the pixels along the way so that you see a line.

We will write our first LogoMation program in just a minute. But before we do, I'd like to mention that pens in LogoMation have a lot of properties. You can have pens of different colors and widths, for example. But let's not worry about that for now. If you don't tell the computer otherwise, it will assume that you want to use a black pen with a width of one pixel, so when you move it around you'll get black lines whose width is one pixel.

And now to our first LogoMation program. We will draw a little rectangle on the screen. How's that for starters? Let me warn you: this is going to be among the most difficult of all the LogoMation lessons, because we have to learn a lot of new things. Once you are done with this lesson, I suggest you let it sink in, at least a day. And I suggest that the next day you quickly review this lesson before going on to the next one. You'll be surprised how easy this lesson is going to be the second time around.

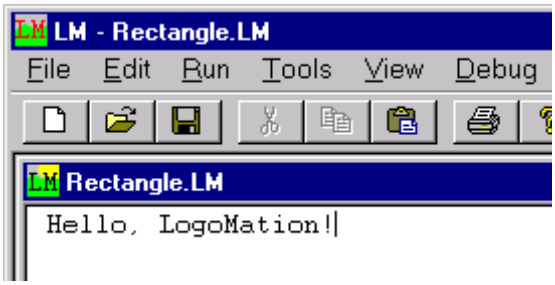
Now, follow me. I suggest that you hold this book next to you when you write your very first LogoMation program. It will help you make sure you do everything as I tell you.

1. Create a directory in which you'll store your LogoMation programs. Call it anything you like. For example, if your name is Charlie, you may want to call it **`Charlie's LogoMation`**.
2. Start LogoMation. Of course, you might need to install it first if it is not installed on your computer yet. And if you are using LogoMation for the first time, you will have to fill in the registration information.
3. LogoMation will open a window in which you are supposed to type your LogoMation program. At this time, the top-left part of the LogoMation window will look more or less like this:



Below the menu line, you have your **`Untitled1`** window, waiting for you to type in something. So, type something, for example, **`Hello, LogoMation!`**
Before writing your first program in this window, it is a good idea to save it under a name that makes sense to you, so that next time you look for the program, you'll be able to find it. How about calling it **`Rectangle`**? In order to give it a name, we save it, just like in most other applications on your computer. We can save it using the **Save** command from the File menu, or by typing control-S – it does not matter. Any way we do it, LogoMation will pop up its standard file dialog window, asking you to select both the file name and the directory it will be in. You type **`Rectangle`** as the file's name, and select the directory you just created in step 1 (if you are

not familiar with how to do this, or do not understand what "directory" is, ask someone for help). Now the screen looks like this:



So far, so good. Now that you have said Hello to LogoMation, let's erase this text. What you typed is not a LogoMation program. We are now ready to type in the *real* program.

4. We will now tell the pen to go straight-ahead 100 pixels. LogoMation starts drawing at the center of the graphics window (you don't see the graphics window yet, but soon you will). The way LogoMation starts drawing, the pen's direction is from left to right, so if you go straight 100 pixels, you will go from the graphics window's center to a point which is 100 pixels to the right of the starting point.
   We tell LogoMation to go 100 pixels by typing **`Forward 100`** as the first line of our program.

5. After going forward 100 pixels, we now want to go up 50 pixels. In order to change the pen's direction, we tell LogoMation to turn the pen's direction left by 90 degrees. So the second line of our program will be **`Left 90`**. Note that when LogoMation changes the pen's direction, nothing happens graphically. All the Left command means is that the next line will be drawn in a different direction.

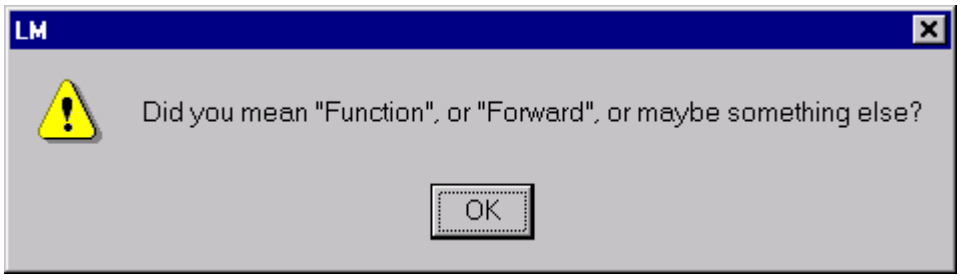
6. You may have guessed the next statement (a statement is just a line in LogoMation in which we tell the computer to do something): it is **`Forward 50`**. Just before the statement is executed, we are at the end point of the previous line, and the direction is facing up. At the end of the **`Forward 50`** statement we have moved up. So the part of the picture which we have so far should be something like a rotated L shape:

   Of course, we still do not have any graphics on the screen. The picture of the rotated L shape is in our mind.

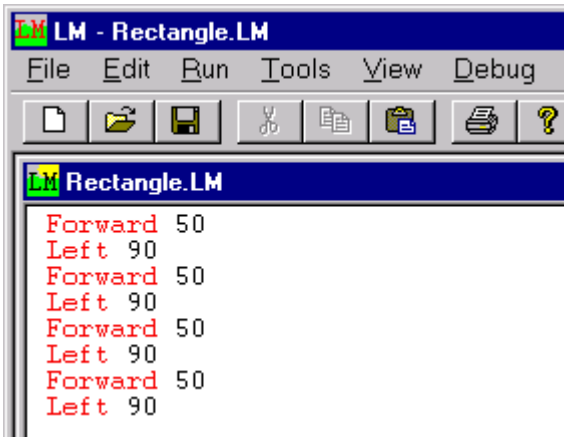
7. If we now turn 90 degrees left and go 100 pixels forward, we will be drawing the third line in our rectangle. So the next two LogoMation statements will be **`Left 90`** and **`Forward 100`** (if you are not sure why, you should read back enough to understand). But wait! Don't type just yet. I want to tell you now about abbreviation. Abbreviation in LogoMation is when you do not type a command name in full. You type just the first few letters. For example, the following are all abbreviations of the Forward command: **`fo`**, **`For`**, **`FORW`**, **`fOrWa`**. Note that I sometimes used upper-case letters, and sometime lower-case. LogoMation does not care which ones you use for command names. So instead of typing **`forward`**, you can just type **`Fo`** and that would be just fine. Likewise, **`Left`** can also be abbreviated, for example, to **`Le`**. However, too much abbreviation can be confusing to LogoMation. For example, a LogoMation statement, which we will learn later, is **`Function`**. Now, **`Function`** and **`Forward`** both start with the same letter, so if we just write **`F`**, LogoMation will not be able to tell if we meant **`Function`** or **`Forward`**. So if you abbreviate the **`Forward`** command too much, LogoMation will complain

about it when it tries to do what you asked it to do. It will highlight the line with the **F** command and pop up a window asking you what exactly did you have in mind:



When that happens, you simply need to type more letters. Actually, most LogoMation statements can be abbreviated down to two letters (there are exceptions to this, though).

8. The last two statements should now be obvious: they are **`Left 90`** and **`Forward 50`**. This will bring the pen back to its initial location. Your screen should now look like this:



By the way, you must have noticed that LogoMation colors the commands. It uses red to highlight the command, and black for the numbers. We will see more colors as we go on.

9. The big moment has come. We are about to run the program. In order to do that, simply press the **GO** button (or you can just as well select the **Go** command from the **Run** menu). What happens next is that LogoMation checks the program for errors, and if none are found (none should be found if you followed the directions above), it opens the Run window and, following your commands, draws the picture.

That was a rather long exercise. Now that you have your first program written, you can enjoy the rectangle, and when you are done, you can get rid of the graphics window, by clicking on its close box.

Now you can try a few experiments of your own:

- Try to change one of the numbers in the program, click GO and see what happens. Make sure you fully understand why the graphics are the way they are.
- Now try replacing one of the **`Left`** statements with a new statement: **`Right`**. It will be fairly easy to understand how that affects the picture.
- Try to replace a **`Forward`** statement with a **`Backward`** and see what happens.
- Next, try to draw a triangle.

- Last, I'd like you to experiment with negative numbers: what happens if you go forward a negative number of pixels? What happens if you turn left a negative number of degrees? Try it, and try to understand what makes LogoMation behave the way it does.

Lesson B: Let's Move Some More

In the first lesson we learned how to use the Forward and Left statements, and if you did your homework, you also know what Right and Backward do. You may have actually noticed that if you use negative numbers, you can get Forward to behave like Backward, or Left to behave like Right. So, for example,

- Forward -100 is the same as Backward 100,
- Forward 100 is the same as Backward -100,
- Left 45 is the same as Right -45, and
- Left -45 is the same as Right 45.

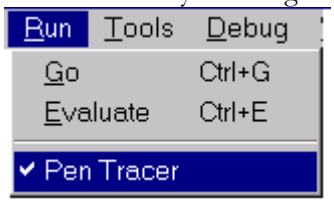
In fact, Left and Right are even closer relatives. Since there are 360 degrees in a circle, if you turn Left 360 then you are really facing the same direction as before, so it is just like doing Left 0, which is really doing nothing. Now, what happens if you turn Left 350? This is almost like completing a 360 degrees circle, except that you stop 10 degrees short of completing the circle. Try to do it yourself: stand up facing a chair, then turn left almost a full circle, but not quite. When you're done, you are now facing a direction that is right of the chair. In fact, if you really turned 350 degrees (which is hard to measure when you are spinning: you get dizzy), then you end up at the same direction you would face if you turned 10 degrees to the right. Likewise, if you want to turn 90 degrees left, you can just as well turn 270 degree right. And in general, if you want to turn x degrees left, you can turn 360-x degrees right. And vice versa. But, enough of that.

Before we continue, I would like to introduce you to the *tracer*.

> Note: the tracer is only available on the Windows version of LogoMation. A related functionality is available on the Mac using "turtle mode", described in Section 12.7, *Tracer Mode*, on page 128. If you are running on a Mac, you can skip the description of the tracer below.

The tracer helps you visualize the movement of the pen. When it is off, drawing is done at maximum speed, which sometimes might look instantaneous. For example, if the tracer was off when you drew the square in the previous lesson, then as soon as you pressed the **GO** button you saw the square. There was no sense of which line was drawn first, and in what direction the pen was moving as it drew the square. Many people are happy to write programs in that way, and that is fine, as long as everything goes according to plan. But sometimes what you get on the screen is not quite what you had intended, and it might not be obvious which erroneous **`Forward`** or **`Right`** is to blame. In such cases, you may want to run things in slow motion. That is precisely what the tracer is for.

The tracer is activated by selecting the **Pen Tracer** command from the **Run** menu:



You will note that there is a little check sign to the left of the menu command. This indicated that the tracer is *on*. If the check mark is not there, the tracer is *off*. Each time you select the **Pen Tracer** command, the check mark is toggled, thus setting the tracer on or off. LogoMation remembers the

setting of the tracer mode between LogoMation sessions. By the way, there is another way to turn the tracer on or off from the program itself, but we will not be talking about that for now.

When the tracer is on, you will see a little **>** sign moving along with the pen. The direction of the sign shows the direction of the pen.

Play a little with the tracer. My advice to you is not to enable the tracer unless you need it, i.e., unless something has gone wrong in your program and after scratching your head, you still don't know what it is.

Now I would like to tell you about four new LogoMation statements. The first is Width. You use this statement to control the width of the line your pen draws. If you don't tell LogoMation otherwise, it uses a pen with a width of 1 pixel. You can change the width, for example by **`Width 10`**, which will draw thick lines.

The next statement I'd like to tell you about is **`Up`**. This is the first statement we encounter that does not need an *argument*. An argument is the number that is part of a statement. For example, in **`Forward 50`**, the **`50`** is the argument. So a statement has two parts: the command (e.g. **`Forward`**), and the arguments. There can be zero or more arguments following the command, depending on the type of statement. Anyway, the **`Up`** statement simply lifts the pen up, so it does not draw when we move it. After all, how many interesting pictures can you draw using a pen which is not allowed to ever be raised off the paper?

Can you guess the next statement? It is Down, which is just the opposite of **`Up`**. If your pen was moving in the Up position then **`Down`** will bring it down so it can draw again.

What if your pen was already up when you used the Up statement? The answer: nothing will happen, and the pen will still stay up. Likewise, if the pen was down and you use a Down statement, it will stay down and no harm will be done.

Using these three new statements, you can draw quite a lot of interesting things. The next statement that I'll tell you about will save you typing. The statement is Repeat, and using it you can tell LogoMation to execute a bunch of statements several times. Let me demonstrate.

Suppose you wanted to draw a square with each side being 100 pixels long, and then to lift the pen up. Here is a program that does it:

```
Forward 100
Left 90
Forward 100
Left 90
Forward 100
Left 90
Forward 100
Left 90
Up
```

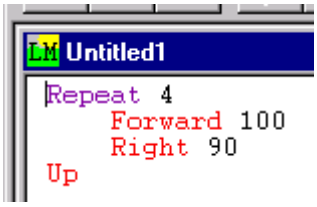
Actually, when I typed this program, I did not really type every one of the nine lines. Instead, I just typed the first two lines, and using the editor's copy and paste menu commands, I replicated them four times. Then I added the last line.
When you use the Repeat statement, LogoMation does this automatically for you. Here is a LogoMation program that draws the same square, but has only four lines:


```
Repeat 4
    Forward 100
    Left 90
Up
```

Now, let's see how it work. When LogoMation sees the **Repeat 4** statement, it knows that it has to repeat the statements that follow four times. But how does it know how many lines to repeat? Simply – by noting that the statements to be repeated (Forward and Left) are indented, that is, they do not start at the beginning of the line. Up till now I did not tell you whether you could use spaces at the beginning of the lines, before the first letter. Now I do: you can. You can start a statement in the middle of the line, if you like, and have any number of spaces between the command and the argument, and in between the arguments. Spaces do not matter much to LogoMation except when it has to *group* statements together. A group of statements should be indented relative to the statement that starts the group. In our example, the Repeat statement starts at the beginning of the line. As long as the statements that follow the Repeat do not start at the beginning of the line, LogoMation knows that they are related to the Repeat. So, LogoMation knows that **Forward** is part of the Repeat group, as well as **Left**. But when it reads the next statement (**Up**), it notice that it is not indented relative to the Repeat, so that must be the end of the Repeat group.

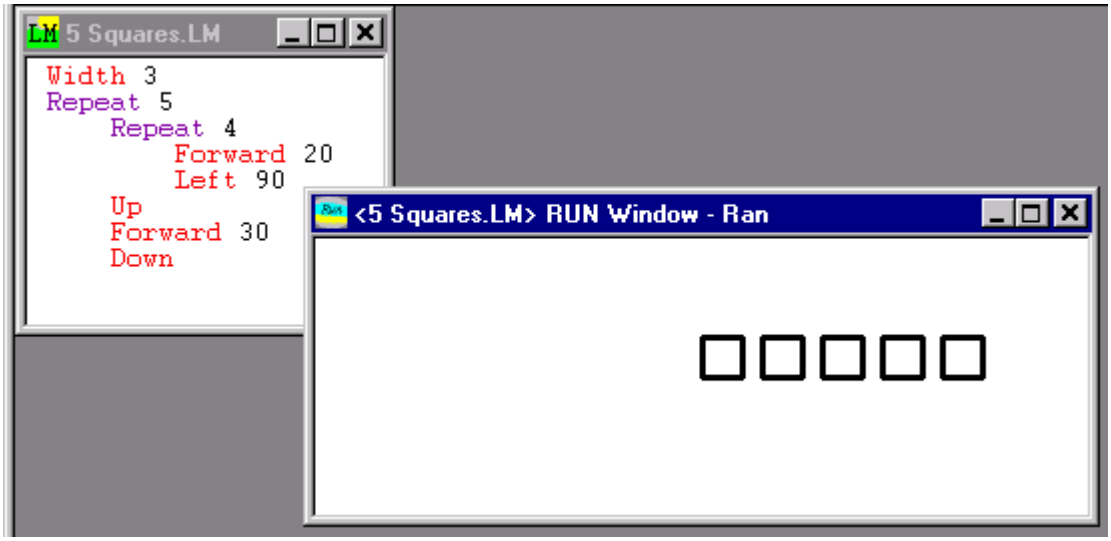
Actually, you may notice that when you type a space at the beginning of the line, you get more than one space – you get four. This is because LogoMation wants to help you *see* the indentation, and sometimes a single space is just not clear enough. And just like a single space typed at the beginning of the line indents four space to the right, deleting one of the spaces at the beginning of a line indents the line four spaces to the left. It also jumps the cursor to the first letter in the line. And last, you may have noticed that if the line you're at is indented, then when you type a return (new line), LogoMation indents the next line the same way the previous one is indented. This is to help you write a long indented paragraph. When you're done indenting, just delete the extra blanks at the beginning of the next line.

Here is what the screen looks like after you type in the program:



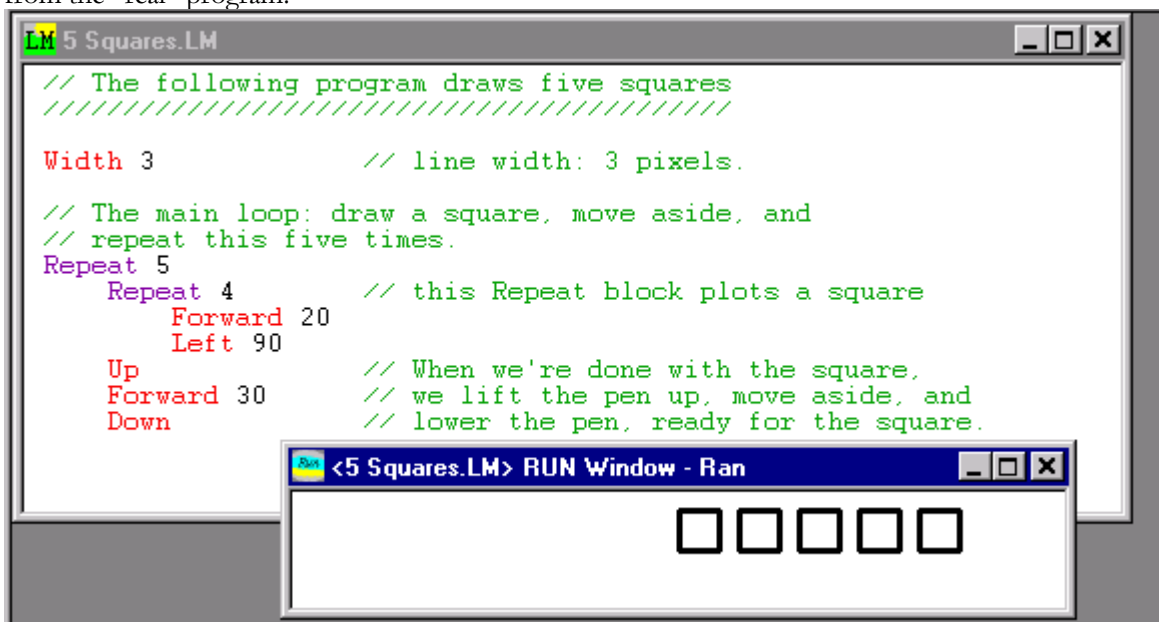
You will note that the Repeat command is purple and not red. LogoMation uses purple to indicate a *compound* statement, that is, a statement that modifies the behavior of other statements that follow it and are indented relative to it.

Do you think we can have a Repeat inside another Repeat? The answer is yes. It is actually very useful. For example, suppose we wanted to draw five little boxes, side by side. Here is a screen shot of the program that does it, and the resulting Run window. Try to type this program in and see if you get the picture below when you run it:



Before ending this lesson, I would like to mention *comments*. Comments are remarks written in English (or any other language that you care to write in). They are totally ignored by LogoMation, and their sole purpose is to let you describe what the program does, in case you will forget the next time you read it. If you never wrote programs before, this may seem like a silly idea: why should you spend the time and write these comments, if LogoMation ignores them? Isn't it true that what the program does is obvious from just reading it? Well, the answer is that what the program does is usually obvious to you when you write the program, but the next time you read it, which could be only a few hours later, you may not remember where exactly does everything belong. This is especially true for larger programs, programs that take tens and even hundreds of lines.

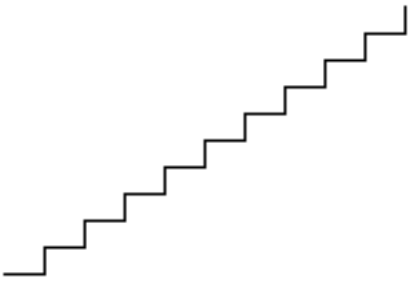
A comment is something that starts with two slashes: // and ends at the end of the line. If you start a line with two slashes, the rest of the line is a comment. Note that in order for LogoMation to be able to tell where a comment starts, you must not put any spaces in between the two slashes. Also, blank lines can be put anywhere, to improve the "looks" of the program. Let's end this lesson with the same program as before, this time with some comments. As you can see, LogoMation colors comments in green. It does it so that they stand out, and also so that they are easily distinguishable from the "real" program:



Does the program look nicer now? You don't have to agree with me, but I think it does. Do you fully understand how the program draws five squares? If not, read it again till you do.

Now for the exercises:

- Draw a staircase, something like the following. By the way, my program had five statements. How many did yours have?



- Draw a black square (hint: first draw a picture like the one on the left, then set the line width such that the lines actually cover the entire area). In the next lesson we will learn easier ways to fill areas.

- Draw three stars:

In order to draw a star, you will have to go Forward, turn Left, and again go forward and left, five times altogether (I hope you are using **`Repeat`** and not just copying the code five times). How much left do you have to turn? Here is one nice feature of LogoMation: you can either think about it, come up up with the number and have it work the first time, or simply try different numbers till you get the right one. I'll give you two hints: one for finding the number of degrees by trial and error, the other for finding the number by thinking.

- Trial and error: Think about the angle between two lines. It is clearly more then 90 degrees, and is clearly less then 180 degrees. Try something in the middle, for example, 160. Write the program using **`Left 160`**, and run it. Does the end of the fifth line touch the beginning of the first? Did you turn left too much, or too little? If too much, try lowering the number, maybe to 130. Is it too little? If it is, it must be between 130 and 160. Continue trying.
- Thinking: imagine yourself at the pen, walking along the lines that make the star. As you go, keep looking at the center of the star. How many times did you have to go around it? Each complete circle around the center is 360 degrees. If you know how many *degrees* you had to turn, and since you know how many *times* you had to turn, the rest is a simple math exercise.

Lesson C: Text, Color, and Fill

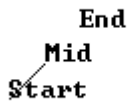
Sometimes it is useful to add some text to a picture. We do it with the Print statement. The Print statement can print numbers or *strings*. A string is any sequence of characters – letters, digits, spaces, and any other character. Strings have to be put inside quotes. The quotes are not printed, they simply help LogoMation to know where is the beginning and where is the end of the string. The following are all valid Print statements:

```
Print 3          // simply print the number three
Print "3"        // the same thing
Print 3.14       // print just that: 3.14
Print "A B C"    // prints the letters A, B and C with spaces in between.
```

Just like in most word processors, LogoMation allows you to change the shape, the size and even the angle of the printed text, but let's not worry about that for now. Needless to say, printing is done at the current pen position. Printing is done even if the pen is up. So the following code:

```
Left 45
Print "Start"
Forward 25
Print "Mid"
Up
Forward 25
Print "End"
```

will display:



Make sure that you understand why this little program works the way it does.

When you print a string, you can put any letter you want inside the quotes, except for quotes. If you really have to print the quote sign, you should precede it with a back-slash, like that:

```
Print "And then he said, \"Quiet!\""
```

which would print

And then he said, "Quiet!"

As I have said, Print displays the text starting at the current pen position. As the example above demonstrates, the pen's position does not change after the printing. Sometimes it is necessary to move the pen to the end of the printed string. This is done by adding a second argument to the Print statement. The argument should be a number. If this number is zero, the pen does not move after printing. Any other number, e.g., 1, will cause the pen to move to the end of the printed string. Example of such a statement:

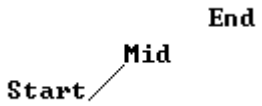
```
Print "two words", 1
```

So, the Print statement is the first statement that we encounter that can have more then one argument. In fact, it is the first statement that we encounter which can have different number of arguments – one and two. There are other statements in LogoMation which can have different number of arguments, and in fact even the familiar Up and Down statements can have different number of arguments.

To actually see what happens if we add the argument to the Print statements, let's add it to the code from the previous example:

```
Left 45
Print "Start", 1
Forward 25
Print "Mid", 1
Up
Forward 25
Print "End", 1
```

we get:



Note how the line starts where "Start" ends. Do not go on reading before you fully understand this example.

Let's turn now to another subject: colors. If you have a B&W (black and white) screen, you would probably not be very interested, but if you have a color display, you're gonna love it. Most computers nowadays have color displays, and I certainly hope you have one. If you do, and you look very closely at the screen, you will be able to see that each pixel is made of a tiny red dot, a tiny green dot, and a tiny blue dot. It might be hard to see on a computer display, but on a TV screen where each pixel is much bigger, you'll be able to see these dots quite clearly[1]. The way your TV, as well as the computer monitor, displays a color in a pixel is by illuminating each of the three colors so that from a distance they look like one dot, with the required color. For example, if the tiny red dot is bright while the green and blue dots are dark, you'll see a red color. If both the red and the green are on and the blue is off, you'll get a yellow. If all three colors are on, you'll get white, and if none is on you'll get black (don't get confused with the way "real" colors, like water colors or crayons, are mixed. If you mix a "real" red color with a green, you will not get a yellow. That is because real colors work by *absorbing* light and letting only light with the right color to go through).

Computer people call the three colors RGB, for Red, Green and Blue. Each of the three colors is specified by its *intensity*, which is a measure of how bright it shines. In LogoMation, a number in the range 0 to 1 specifies the intensity, where 0 is the lowest intensity (color is off), and 1 is the brightest it can get. For example, 0.5 would be half intensity.
The way we tell LogoMation to change the color of the pen is with the Color statement, which requires three arguments: red, green and blue. Here are some examples:

```
Color 0,0,0     // black
Color 1,1,1     // white
Color 1, 0, 0   // bright red
Color 0, 0, 1   // bright blue
Color 0.3, 0, 0 // darkish red
```

---

[1] Note: in some Sony displays you will see stripes of red, green and blue, as opposed to little dots.

At this point you may wonder: how am I ever going to tell LogoMation to display the color I have in mind? For example, what is the RGB of light brown? What is the RGB of Navy blue or dark purple or light pink? Luckily, LogoMation provides a *color picker*. All you have to do is to position the cursor where you want the RGB colors to be typed, and then use the **Select a Color** command from the **Tools** menu. Doing so will pop up the color picker, and you will be able to pick the color of your choice. When you picker the color, click OK, and LogoMation will type the RGB numbers for you. If you want to change a color, simply select the RGB numbers in the editor and invoke the color picker. Note that you should select only the three numbers – not any other part of the Color statement. The color picker now would display both the old color and the new one, allowing you to compare the two.

You can switch colors any time in a LogoMation program. Every line which the pen draws after a color change will use the new color — unless you change the color again.

While we are using the **Tools** menu, maybe it is time to learn about three other time-saving tools, which are available in the same menu. These are **Beautify**, **Indent Left** and **Indent Right**.

Beautify reads your program, and converts commands to capitalized words, fully spelled out. So if you had an **`LE`** command, for example, then **Beautify** will convert it to **`Left`**. You can also use ctrl-B to achieve the same thing (note: on Macintosh computers, you use option-B instead of ctrl-B. The same is true for other control sequences). The **Beautify** menu command works on the selected lines or on the current line if nothing is selected. A common practice is to select all the lines in the buffer before beautifying (selecting all lines is done either by dragging the mouse over them, or better, by using the **Select All** menu command, ctrl-A for short).

The **Indent Left** and **Indent Right** menu commands help you indent a entire region right or left. If nothing is selected, **Indent Right** indents the current line, plus select it. You can use ctrl-] as a shortcut for **Indent Right**. Likewise, **Indent Left** removes blanks (if any) at the beginning of the line. You can use ctrl-[as a shortcut for **Indent Left**. If several lines are selected, the indentation menu commands indent the entire block of lines and select it. The selecting of the lines after the indentation helps if you want to indent some more. For example, you can type two ctrl-] and indent right by 8 blanks.

If you did your homework for Lesson 2, you know that it is possible to fill a shape by drawing inside it using a thick pen. But for anything other than rectangles, this is really pretty hard. We are now going to learn about the Fill statement.

The Fill statement tells LogoMation to fill the stuff inside a closed path that the pen makes. Like Repeat, we use indentation in order to tell LogoMation which pen movement statements are included in the area which should be filled. Here is an example:

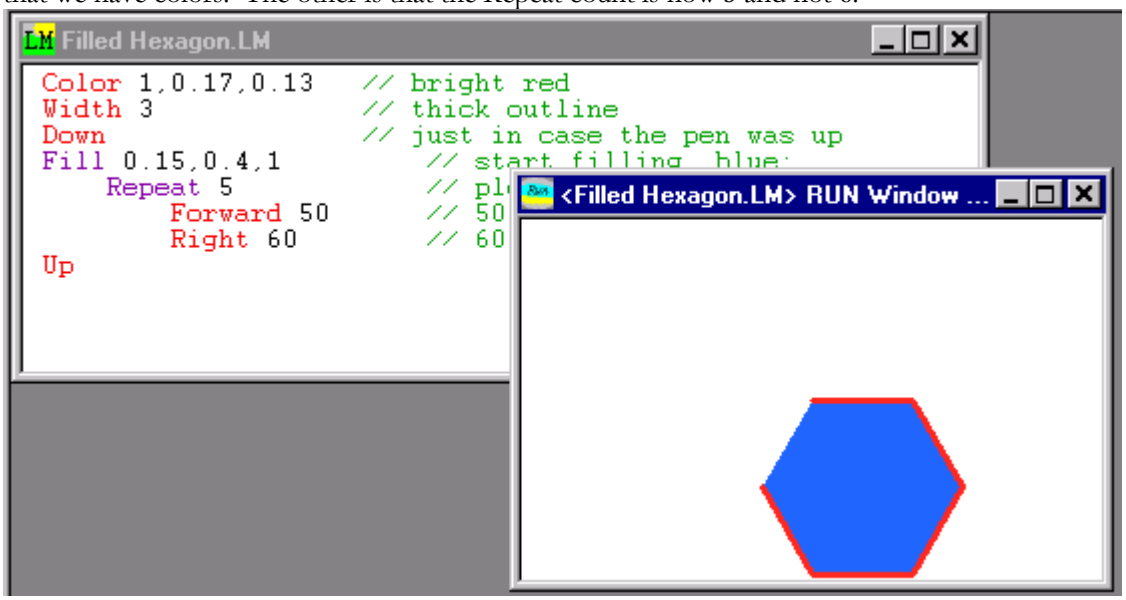
```
Down                // just in case the pen was up
Fill                // we start filling:
    Repeat 6        // ready to plot a hexagon -
        Forward 50  // by drawing 20 pixels lines
        Right 60    // with 60 degrees between them
Up                  // done Repeat, done Fill
```

When we run the program, we get

To understand why we got what we got, try to comment out the Fill statement (commenting out means typing // at the beginning of the line, so that LogoMation will ignore the entire line). What did you get? Well, the Fill statement simply fills that shape.

The filling that we have just seen was *solid* fill. LogoMation also supports filling with other patterns, such as round dots or crisscross lines, but we will not talk about it in this tutorial. The Reference Manual part of this book describes filling with patterns (sections 5.3.4 and 5.13). Not that it's difficult to do – but I'd like to defer the topic of patterns so that we can cover other important features.

How about filling with colors? Well, since we know how to change the color of the pen, we can draw filled shapes with these colors. The Fill statement, which we have just seen, fills a shape with the same color as the pen's. So if we wanted a red hexagon, for example, all we had to do was change the color to red before using the Fill statement in the last example. But LogoMation can do even better. You can tell LogoMation to draw the outline in one color (which is the color of the pen), and do the filling in another color. You tell LogoMation about the color of the filled area by adding three arguments to the Fill statement. The three arguments are our old friends RGB (remember, you can use the color picker to help you type them in). We end this chapter with yet another hexagon. It is almost the same hexagon as the before, but there are two differences. One is that we have colors. The other is that the Repeat count is now 5 and not 6:



How come we got a hexagon even though the repeat count was only 5? Remember, the Fill statement works on closed paths. A closed path is a path that the pen makes, which ends at the point in which it started. What if we do not close the path inside the Fill statement? In that case, LogoMation will close the path just for filling purposes, but will not draw the extra outline. That is why we got a filled hexagon even though we actually drew only 5 of the 6 edges.

Before giving you this lesson's homework, I'd like make a confession. The hint that I gave you in the homework of the last class, on how to draw a black square, was a bit misleading. What I wanted you to do was to exercise the use of the Width and the Repeat statements. But if you chose to ignore my hint, you could instead write a two-line program, with no Repeat, that does the job. Did you think of it yourself? I hope so. What lesson does it teach you? That you should not believe books too much, because they can sometimes be wrong, too.

Now for the homework:

- Draw three ten-sided polygons side by side, each with a green outline and blue filled inside.
- Modify the three stars exercise from the last lesson, so that the stars are inside a Fill block. So far, easy enough. Now for the challenge: can you figure out why you got the picture that you got? You probably got something like:

  This exercise teaches you something about filling a shape made of lines that cross one another. Try some more such exercises: draw a shape with lines that cross one another, then indent the program (ctrl-] might be handy) and put a Fill statement in front of it. After a while, you'll understand why LogoMation behaves the way it does.
- Draw a STOP sign. It should look like the one in the street around the corner, or maybe two blocks down the road. Use colors. Later, when you know more about fonts, you'll be able to change the size and shape of the STOP letters. By the way, have you figured out how to change the color of the STOP text?

Lesson D: Circles

This lesson is going to be short. I want to give you a little break, and I want you to spend more time doing the exercises and inventing some of your own.

We begin this lesson by learning how to draw circles, or part of circles (arcs). We will have to learn two new statements. The first statement is Straight. You use this statement to tell LogoMation that all the Forward and Backward statements that will come next are to be done in straight lines. In fact, all the programs that we wrote thus far had only used straight lines, so you could use this statement anywhere in a program without changing anything in the way the program works. Here are two programs that do exactly the same thing (draw a square), one with **`Straight`** and one without.

```
Straight
Repeat 4
    Forward 30
    Right 90
```

```
Repeat 4
    Forward 30
       Right 90
```

We tell LogoMation to move in arcs by using the Circle statement, with the circle's radius being the argument (I will tell you how to do it in the next paragraph). When we draw a circle or part of a circle, the Forward and Backward statements will actually move along a circular path. When we are done moving in circles, we use the Straight statement to tell LogoMation that Forward and Backward will now mean what they used to mean when the program started, namely, that we want to move in straight lines.

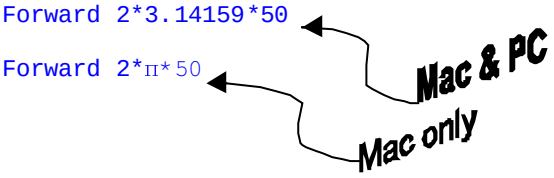
A circle has a radius – you probably know this already. You tell LogoMation that you want to move on a circle with a radius of say, 50. Then you use a movement command such as a Forward statement. LogoMation will start going in the same direction it used to go, except that it will curve, and if we go far enough we will get back to where we were when we started the circle. Here in an example, which includes the code and the resulting graphics:

```
Forward 40
Left 45
Forward 40
Circle 50
Forward 100
```

What have we done here? We started by drawing a horizontal line of length 40, then we turned left 45 degrees and drew a second line. We then told LogoMation to move in a circle with a 50-pixel radius. The last statement was to move forwards 100 pixels. This last statement was executed when LogoMation was in circle mode. It starts from the end point of the previous Forward, and curls instead of going straight. How much do we have to move forward in order to get to where we started? The answer is: 2 times π times the radius, which in our case is 50. If you don't know about this magical number, π (pronounced Pi), ask someone.
So, to get a complete circle, you can use a calculator, figure out how much is 2 times π (π is approximately 3.14159) times 50, and enter that number as the argument of the Forward statement.

Actually, LogoMation can help in doing the math. In the next chapter we will talk a lot more about how to do math in LogoMation. For now, I will just mention that instead of using a calculator and copying the result of the multiplication, you can simply type 2*3.14159*50. Actually, on the Macintosh you can do much better – instead of typing the value of π, you can simply type π by holding down the option key and pressing lower-case p. So the Forward statement would look like this:

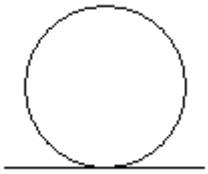
```
Forward 2*3.14159*50

Forward 2*π*50
```



So let us write a simple program that draws a straight line, tangent (this means, touching at one point) to a circle. The program:

```
Forward 50
Circle 40
Forward 2*3.14159*40
Straight
Forward 50
```

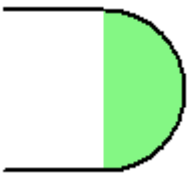
And here is what it does. Note that the straight line at the bottom is really made of two line segments, each 50 pixels long. You should thoroughly understand the program before you go on.



Of course, circles, or part of circles, can be filled too. Here is a slight modification to the program above:

```
Width 2
Forward 50
Fill 0.5,0.97,0.53
    Circle 40
    Forward 3.14159*40
Straight
Forward 50
```

And here is what it does:



Now for the homework:

- Draw a car, with wheels (how else would it move?). Make it as realistic as you like, but filling it with lively colors. Do not forget to save the picture in a file!
- Draw the logo of the Olympic Games. By the way, do you know why this logo is the way it is? I knew once, but now as I write this book, I forget.
- Make up your own pictures, with colors, lines of different width, filled shapes and circles.

Lesson E: Variables and Functions, Part I

What we have learned thus far had enabled us to do simple things, like drawing shapes and filling them with colors. In this chapter we will learn about variables and about functions. Variables and functions are very powerful: they allow us to write programs that do complex things, and yet these programs can be kept simple, easy to write and easy to understand. If you saw a demo of LogoMation in which LogoMation draws real nice pictures, you can bet that these programs used variables and functions. By the time you finish this lesson and the next one, you will be able to do some of it yourself. But be warned: if LogoMation is your first programming language, learning about variables and functions may not be very easy. Take your time and study them carefully.

People use functions (also called "procedures" in other programming languages) for different reasons, but by far the most frequent use of functions is when you want to do the same thing again and again in your program. You may remember that we used Repeat to avoid replicating the same lines many times. Well, functions are even more powerful and general. Let's start with an example. We will write a program for drawing a few squares of different sizes on the screen. We will first write if without using a function and then see how using a function helps in getting the program shorter and simpler. Here is the program.

```
// Move a little, with the pen up
Up
Forward 30
Down
// Draw a 5 x 5 square
Repeat 4
    Forward 5
    Right 90

// Move a little, with the pen up
Up
Forward 30
Down
// Draw a 10 x 10 square
Repeat 4
    Forward 10
    Right 90

// Move a little, with the pen up
Up
Forward 30
Down
// Draw a 20 x 20 square
Repeat 4
    Forward 20
    Right 90
```

It is not too difficult to understand what this program does, but please try to do it before you continue reading. Got it? It is really a simple program: it moves 30 pixels to the right, then draws a 5 by 5 pixel square, then moves 30 again and draws a 10 x 10, and then does this yet another time and draws a 20 x 20 square. I suggest you type it in and run it, just to see how it works.

If you read the program a second time, chances are that you will find it boring. After all, it does almost the exact same thing three times. The only thing that is different each time is the size of the square – just one number. Surely there is a way to write the program without having to replicate the code and modify it slightly in every copy.

This, by the way, is what every good programmer is always trying to do: write programs that are smaller and clearer. A small program is easier to write, easier to debug (that is – to find the programming errors, also called "bugs"), and many times it is more efficient, i.e., it runs faster.

If you read the program and think about how to make it shorter, you may think of using a Repeat statement to reduce its size. Unfortunately, a Repeat, at least the way we learned about it in Lesson B, cannot help. This is because a Repeat executes the same statements again and again, each time exactly the way it did before. It is this *exactly* that we don't like here, because each square is slightly different in size. So instead, we will use a Function. Here is the program, re-written with a function:

```
// square – define a function to draw a square
Function square(size)
    Up
    Forward 30
    Down
    Repeat 4
        Forward size
        Right 90

// now draw three squares
square(5)  // draw a  5 x  5 square
square(10) // draw a 10 x 10 square
square(20) // draw a 20 x 20 square
```

OK, so what have we got here? Everything seems new, and yet some lines look familiar. Let's read it together. The first line is a comment, which is always a good idea. The comment says that we are defining a function. A function is something that you *define*, which is something that we did not see before. When we define a function, we just tell LogoMation that if we ever use this function, this is what it should do. But we are not asking LogoMation to do anything yet. Programmers also call this "declaration", because we declare something to be such and such.

The second line is the actual beginning of the function. As usual in LogoMation, we use indentation to define the scope of the function, i.e., where it begins and where it ends. So, our function has seven lines altogether – the line that starts with **`Function`**, and six additional lines. The line that starts with **`Function`** has the name of the function, **`square`** in this case. The function's name is entirely up to us. It is better to use a meaningful name to tell something about what the function does (draws a square in our case), but it is really up to us, and we can pick any name we like. The only restrictions on names are that they must be made of only letters, numbers, and underscores ("_" character), and must start with a letter.

Following the name **`square`** we find the word **`size`** in parentheses. This tells LogoMation that whenever the function is *called* , the call should provide an argument. *Calling* a function is the term programmers use to describe the act of actually using the function to do something -- more on that later. Again, the name of the argument is up to us. I chose **`size`** because I want the argument to tell LogoMation the size of the square. Indeed, if you look at the last three lines of the program, each of them is a call to the **`square`** function, and each has a size number in parentheses. But let us go back to the function's declaration.

Each function declaration has two parts: the Function line, also called the function's *head*, and the function's *body*. The function's head is where we tell LogoMation how we will later call the function. The function's body is where we tell LogoMation what the function should do once we call it. Read the square function's body. It has six lines, all of which you have seen before, at the beginning of this chapter. There is a Repeat, and a Forward, and other statements. Look carefully. Do you see something new? Read again. The new thing is the following line:

```
Forward size
```

When LogoMation sees this, it knows that we want it to go forward. What is the Forward statement's argument? It is **`size`**. We call **`size`** a *variable*. A variable is something that has a name (**`size`** in this case), and also has a value. What is the value of **`size`**? In this case, since we used **`size`** in the function's declaration, we are actually telling LogoMation to use the value that will be supplied in the function's call.

Confused? I don't blame you. This is all new stuff, as I said before. It is really simple, once you digest all these new words. It's one of those cases where the explanation is more difficult than the concepts it is trying to explain. I am now going to tell you what really happens when you call a function, and hopefully it will all become clear.

The first function call in our program is this:

```
square(5)
```

This is common to all function calls. All function calls have a word, which is the function's name, followed by zero or more arguments, all in parentheses. In this case there is one argument, but if it had more, we would have to separate the arguments with commas. By the way, no spaces are allowed between the name and the left parentheses. Likewise, no spaces are allowed between the name and the left parentheses in the function's head. This is one exception to the rule that you can add spaces in a LogoMation program anywhere you want, to clarify the program and make it nicer.

Here is an example of another function call (not in our program):

```
rectangle(10, 6)
```

This call calls a **`rectangle`** function and passes it two arguments, 10 and 6. Note again that there is no space after the function's name. But there is a space after the comma. Why? Because I think it looks nicer. You don't *need* the space.

What happens when LogoMation runs the program and gets to the **`square(5)`** function call? It does the following:

1. It looks for a function called **`square`**. In this case, we just defined it before the call, so it finds it without any problem. If we had not defined it, LogoMation would complain. By the way, you can define the function anywhere in the program, not necessarily at the beginning. You can even define it after the first call, if you want. LogoMation reads the entire file before executing the first statement, so it does not matter to LogoMation where you put the function definitions.
2. From the function's definition, LogoMation understands that there is one argument, **`size`**. It checks if the call also has one argument, which it does. LogoMation then sets a variable called **`size`** to the value supplied by the call, i.e. to 5. So from now on, till the end of the function, every time LogoMation sees the word **`size`** in the function's body, it replaces it with 5.
3. LogoMation starts executing the function. This means that it reads each statement in the function's body and executes it. So, it will do Up, and then it will do Forward 30, and so on. When it gets to the **`Forward size`** statement, it will do Forward 5, because the variable **`size`** has the value of 5.
4. After executing the function's body, LogoMation *returns* from the function. This means that it goes back to the next statement following the function's call. In this case, it returns from the **`square(5)`** call, and is ready to execute the next statement, which in this case is also a

function call. When LogoMation returns from a function, it "forgets" the definition of the arguments (**`size`** in out case); the definition of function arguments holds only during the execution of the function.

If you are still a bit uncertain what is going on here, please read the explanation again until you fully understand. It is important that you fully understand what it means to *define* a function, what is the function's *head* and *body*, what is a function *call*, and what it means to *return* from a function.

There is more to be said about functions and variables, but we will break now.

Now for the homework:

- Write a function **`polygon(n, size)`** which will draw a polygon with **`n`** sides, with each side length being **`size`**. Use the function to draw 4, 5, 6, 7, and 8 edged polygons.
- Using a function with two arguments, draw 5 rectangles separated from one another by 10 pixels. The sizes should be 4x7, 6x3, 10x11, and 20x20.
- Modify the above program so that the 10x11 rectangle will be filled with solid black, while the rest of the rectangles remain not filled.
- Modify the above function yet another time, so that all the rectangles will be filled with solid black.
- The next exercise is going to be tricky – but I'd like you to try to do it anyway. Start with the program in this lesson, the one with the three squares. I'd like you to modify it by adding a second argument to the function: when the argument is 0, the function will draw a square just like it did before, taking the edge size from the first argument. If it is non-zero, it will draw a circle with a radius specified in the first argument. Call the function, and draw three squares and two circles.
  To write this, you will need to use an **`If`** statement – something that we have not learn about yet. I'd like you to read about it in the Reference Manual section of this book, section 5.11, *LogoMation Conditionals – the `IF...` and `IF...ELSE...` Statements,* on page 66. There are some new concepts mentioned there, but you do not have to understand all of them (for now…). Just try to understand how to write a simple If statement.

Lesson F: Variables and Functions, Part II

Before we go on, I'd like to talk a little about the last exercise in the last chapter. I hope you managed to do it, or that at least you tried. The reason that it is so important is that sooner or later you will have to start using the Reference Manual part of this book. The reference manual describes every feature and capability of LogoMation, and is your main source of information about how to use the programming language and the development environment. The problem is, that the manual is kind of hard to read because it is somewhat concise, and sometimes uses a technical language, which experienced programmers understand, but some less experienced programmers might find confusing. Frankly, if LogoMation is your first programming language, there is little hope that you will understand *everything* in the reference manual. You will have to get someone to teach you advanced programming. But whenever you learn a new concept, or even a new statement type, try to read about it in the Reference Manual part of this book, and thus become more familiar with the way programming languages are described. This ability will become handy when you learn your next programming language.

Back to variables. In the previous chapter we learned about variables. We learned that they can store values. We learned that they can store different values at different times – for example, the **`size`** variable in the example from the last chapter stored a different value each time the function was called. We also learned a way to assign a value to a variable: it was done by the act of *calling* a function. In calling a function, LogoMation assigns the value of the argument (or arguments, if there are more than one) to the variable (or variables, if more than one) in the Function declaration.

There are other ways in LogoMation to assign a value to a variable. In this chapter we'll learn two new ways. The first and the most important is the *assignment statement*.

Here is an example of an assignment statement:

```
G = 10
```

The statement has three parts:

1. The variable's name – in our example it is **`G`**. By the way, I have not mentioned it yet, but variables' names are *case sensitive*. "Case Sensitivity" is a big word that simply means that it matters whether you use lower case or upper case. Thus, for example, the variables **`size`** and **`Size`** are two entirely different variables, even though the only difference in their names is that one is capitalized and other is not. So be careful and consistent with how you type variables! By the way, the same is also true of function names, so **`square`** and **`Square`** are entirely different.
2. The second part is the equals sign (**`=`**). The equals sign tells LogoMation that we have an assignment statement here. The assignment statement is kind of unique, because all the other LogoMation statements begin with a command that tells LogoMation what to do, for example Forward, and the command is followed by the arguments. The assignment statement begins with a variable's name, and only the second part tells LogoMation what to do. Why? Because it is a common practice in many other programming languages. It has become a tradition.
3. The third part is the value, to be assigned to the variable. In our case, it is 10, and thus when you use the variable G after the assignment statement, it is entirely equivalent to using the number 10. The variable will hold its value until you replace it with a different value.

Here is a quick example:

```
G = 10          // set variable "G" to 10
Forward G       // go forward 10 pixels
Right 90
G = G + 1       // increment G by one, to 11
Forward G       // go forward 11 pixels
Right 90
G = G + 1
Forward G       // go forward 12 pixels
Right 90
G = G + 1
Forward G       // go forward 13 pixels
```

The first assignment statement is clear enough (especially since we discussed it so much already). The second statement should also be clear enough: since G equals 10, going forward "G" is the same as going forward 10. The fourth statement is kind of strange, though. Let's stop for a minute and talk about it.

The statement **`G = G + 1`** is an assignment statement. Every assignment statement has three parts (have I said it before?), and the first two parts are just what we had before: the variable's name and the equals sign. The third part, the value part, is **`G + 1`**. Since **`G`** equals 10, the value of **`G + 1`** is 11, and therefore what we actually do is assign **`G`** the value of 11. The confusing part is that we use **`G`** both as the variable to be assigned a value, and as the variable to provide the value. When LogoMation executes an assignment statement, two things happen, in that order:

1. LogoMation computes the value part (the third part of the statement). In some cases there is not much computing to do because the value is simply a number, such as 10. In other cases, some computation needs to be done, because the value has operators such as **`+`** (plus), **`-`** (minus), **`*`** (multiply), and **`/`** (divide). If the value part has variables, LogoMation substitutes their value in the computation.
2. LogoMation assigns the calculated value to the variable whose name appears in the first part.

So, a statement like **`G = G + 1`** should not be confusing at all because there is a clear order: first come the computation and *then* come the assignment. During the computation, **`G`** has not been assigned a new value yet, and its value is 10, so that is the value which is being used to compute the new value.

Personally, I was a bit confused when I learned my first programming language and I first saw a statement such as **`G = G + 1`**, because such a statement can never be true mathematically. In Algebra, the equation **`G = G + 1`** has no solutions: there is no number **`G`** such that **`G = G + 1`**. But in programming, the assignment statement just tells the computer what to do, and does not represent a mathematical equation. You should read it as follows: **`G = G + 1`** means "assign a *new* value to **`G`**, the value being equal to the *current* value of **`G`** plus one".

Why would anyone want to change the value of a variable? Actually, it is extremely useful. The following program will demonstrate:

```
// A spiraling square
////////////////////
side = 10
Repeat 150
    Repeat 4
        Forward side
        side = side + 1
        Right 90
```


Before you continue reading, stop for a minute and try to guess what the program does. It's not at all obvious! When you have clue (or when you give up), continue reading.

OK. Let's read it together. We will start with the inner loop. Oops – I have used another programming jargon! When a program has a loop (i.e. a Repeat statement) within another loop, we say that we have *nested* loops. And when we have nested loops, the *inner* loop is the one that is "deepest", the one that is executed more times. So in our case, the inner loop is the one with the **`Repeat 4`** statement.

We have seen loops like it before, have we not? If we took the **`side = side + 1`** assignment statement out of the loop, it is a familiar loop that draws a square. So, leaving the assignment statement out for one more minute, we have an outer loop that runs 150 times, and each time draws a square. And all the squares are the same, so if we ran the program without the above assignment statement, we would only see one square, although LogoMation re-draws it 150 times.

So now let us add the assignment statement. What is the size of the square now? The first edge is 10 pixels. The second is 11, the third is 12 and the fourth is 13. But wait! This cannot be a square, because the shape is not closed! And as we continue the outer loop, we draw another "almost square", this time with edge sizes of 14, 15, 16 and 17. By now you should have a picture in your mind what this program does. Run it and enjoy! By the way, you may need to adjust the size of the Run window so that it displays all the graphics.

I promised two new ways of assigning a value to a variable. The second way is with a new form of the familiar Repeat statement. This new form loops, just like a normal Repeat, but each time through the loop it increments a variable. With all that we've learned about variables, the following program should be easy enough to understand – especially if I tell you that it does exactly what the previous one did:

```
// Another spiraling square
Repeat size, 10, 610
    Forward size
    Right 90
```

The Repeat statement has three arguments:

1. The variable's name — in our example it is **`size`**.
2. The initial value, to which the variable is set before the loop.
3. The last value – the Repeat loop increments the variable by one every loop, until its value equals the last value. After the variable reach this last value, the loop ends.

As is the case for all of the statements that we have learned thus far, there is more to be said about the Repeat statement. It is covered in the Reference Manual section of this book.

Now that we know a lot about variables, I'd like to mention another programmers' jargon word: an *expression*. An expression is simply something that has a value. For example, 10 is an expression. In LogoMation we use expressions to provide values to arguments of statements, and to arguments of functions. For example, the Forward statement has one argument, and that argument can be any expression, including any math operations that you want, for example

```
Forward ((x1 + 8) * (x1-2)) / y
```


As in math, parenthesis can be used to tell LogoMation in what order to evaluate the expression. Without parentheses, LogoMation computes first the multiplication and division, then the addition and subtractions. So, for example, **`10+20*2`** equals 50, while **`(10+20)*2`** equals 60.

There are four types of "values" in LogoMation. In these lessons we will only learn about two: numbers, and strings. We have already seen both. LogoMation allows you to use them interchangeably – for example, the string **`"10"`** has the same effect in a Forward statement as the number **`10`**. LogoMation will convert a string to a number or a number to a string as needed by the statement. You may want to read the examples and an in-depth explanation in the reference manual, Chapter 2: *Data Types, Variables and Expressions*, on page 46.

Homework (have you noticed, they become harder and harder…):

- Write a program to draw concentric circles.
- Modify the above program so that the circles have different colors. For example, you may want all the circles to have shades of red: just change the intensity of the red while keeping the green and blue intensities at zero.
- Here is a problem that will keep you thinking for a while, unless you have already seen its solution: can you draw a circle without using the Circle statement, i.e., with only straight lines?
- Modify the previous program, the one that draws circles without a Circle statement, and draw one big spiraling shape that would fill up the entire screen. It is essentially a round relative of the Spiraling Square program, which we have seen earlier in this lesson.
- Read and understand the **`Font Test`** program in the **`examples`** directory. You will need to consult the reference manual.
- Read in the Reference Manual about built-in functions, and modify the spiraling circle program to draw the shape until it touches an edge of the Run window. The shape should start at the center of the Run window, and grow outwards. When it touches the edge of the window (either the top of the left edge, depending on the shape of the window), the program should stop. You will probably use the **`x()`**, **`y()`**, **`width()`** and **`height()`** built-in functions, the **`If`** statement, and the **`Break`** statement.

Lesson G: Fun with Animation

You now know enough to start enjoying animation. Animation in LogoMation is done by attaching a *picture* to the pen. Computer professionals call a picture a *bitmap*. What is there in a bitmap? Actually, you do not really need to know this in order to do animation in LogoMation, but I will tell you anyway, just so you do know. If you wish, you can skip this description.

A bitmap has four characteristics that distinguish it from other bitmaps:

1. Size – a bitmap is rectangular. An *n* x *m* bitmap has *n* rows, each with *m* pixels. Alternatively you could say that the bitmap has *m* columns of *n* pixels each, but programmers prefer to talk about rows of pixels. The reason for that has to do with the way that the bitmap is stored in memory or on a disk: it is stored in rows: first all the pixels of the first row, then all the pixels of the second, and so on.
2. Depth – this is the number of bits which are used to represent each pixel. For example, if a bitmap has a depth of 8, then each pixel is represented by 8 bits and therefore can represent 2^8=256 different colors. So what would the depth of a black and white bitmap be? You guessed it: 1.
3. Color table, also called a Lookup Table (LUT), also called a *palette* – this is a table, which translates the value of the pixel (expressed by *depth* bits), to a color displayed on the monitor. As you already know, a color has three components (RGB), and typically each component is represented in the computer by an integer in the range 0 to 255 (do not confuse this "low level" representation with the way LogoMation represents intensities – each in the range 0 to 1). A modern monitor can thus represent about 16 million colors (2^{24} to be exact). The palette maps the value of each pixel to a color. The most important reason that people use palettes is to save memory, because each pixel can have a depth less than 24. There are other reasons, too.
4. And the fourth item? You guessed it – the actual value of each of the *n* x *m* pixels.

When I think about it, I discover that I am not sure why a bitmap is called a bitmap – I am not sure what "bit" has to do with it, and what "map" has to do with it. But anyway, that's the term that programmers use, so we better stick to it.

Now that I told you about bitmaps, just archive the information. We are not going to need it for what follows.

In LogoMation, a picture is an entity that has a unique name. The name of a picture can consist of any character except parentheses (parentheses are used for multi-frame animation, which is described in section 9.5, *Multi Frame Pictures*, on page 111). For example, **`"a picture"`** is a legal picture name. Pictures come into being in two ways: either they are imported to LogoMation from other graphics tools (using the **Import a Picture** menu command), or they are created in LogoMation by standard LogoMation statements.

Can you guess how to create a picture in LogoMation? As you might expect, it is consistent with other statements like Fill or Repeat. You use a Picture statement, followed by movement commands which are indented relative to the statement. For example, a picture "square" can be created thus:

```
Picture "square"
    Repeat 4
        Forward 40
        Right 90
```


Looks simple enough. I would like you to note, though, that the Picture statement *creates* a picture, and does not define it. In other words, the statements in the Picture statement's body must be executed. Contrast this to the Function statement, which is a declaration. You do not execute a function definition – you just put it somewhere in the file. But a picture needs to be created.

When a Picture statement is executed, nothing happens on the screen. All the graphics is stored into a – are you ready? – a bitmap. The bitmap is stored by LogoMation and is brought into use when it is attached to a pen. Attaching a picture to a pen is done via a Pen statement, as in the following example:

```
Pen picture="square"
```

The Pen statement modifies several of the pen's parameters. As always, you can read about it in the Reference Manual section of this book. In this lesson I will only mention another parameter of the pen: speed. When in animation mode, the speed can be controlled by setting its value in the Pen statement. For example:

```
Pen speed=100
```

This tells LogoMation to move the picture at a speed of 100 pixels per second.

Combining all this, here is our first animation: we move a square 200 pixels to the right, and this movement takes 2 exactly seconds. Note that I am setting both the picture's name and its speed in one Pen statement. You can put as many Pen parameters in one statement, or separate them into several statements.

```
Picture "square"
    Fill 0,0,0 // black
        Repeat 4
            Forward 40
            Right 90

Pen picture="square", speed=100
Forward 200
```

Please type this in and run it to see how it works.

Now change the size of the square from 40 to 100 and re-run. Did you get what you expected? I think not. You got the same 40 x 40 square as before. The reason for that is that LogoMation *caches* pictures' definitions. Caching means that the bitmap is saved, and when the program re-runs, LogoMation notices that the picture was already created before and skips its re-creation. This is done to save time, of course, but in our case it is a mixed blessing, because LogoMation skips the new definition of the picture.

Disabling the caching is easy: you simply add an argument 1 to the Picture statement. So the modified program with 100 x 100 square is as follows:

```
Picture "square", 1
    Fill 0,0,0 // black
        Repeat 4
            Forward 100
            Right 90

Pen picture="square", speed=100
Forward 200
```

And now, to the homework:

- Animate a sequence. Define several pictures of the same shape, say, a square, which differ in their name (of course) and in their color. Then move them along a straight line,

alternating between them as you go. For example, you could move the red square for 10 pixels, then switch to the blue and move it for 10 pixels, and so on.

- Animate a bouncing ball. The ball starts at a given angle, and when it hits an edge of the Run window, it bounces back using the opposite angle. To add to the effect, use sound, and increase the speed of the ball in each loop (looks like the ball receives energy from the Run window's edges).
- Do not drink and drive! Take your car drawing program (from Lesson D), combine it with the Stop sign program (from Lesson C), and create an animation of the poor drunken driver hitting the Stop sign. I have been told that typically male student like this exercise much more than female student. I wonder why. Anyway, if you prefer, you could instead animate a peaceful scene in which the driver drives carefully to the Stop sign and stops.
- Add sound effects to the above animation. Use the Sound statement, described in the Reference Manual. Look for cool sounds on your computer, and import one or two of them to LogoMation.
- Animate a square and a triangle moving towards one another. The pictures should look like they are moving concurrently, although in reality you will alternate between them. The most efficient way to do this is to use *named* pens. Read about it in the Reference Manual section 8.2, *Defining and Switching Pens*, on page 103.
- Animate an explosion: a round ball moves on the screen, and then explodes into 10 little balls that are dispersed all over.

Congratulations!

If you have read and understood the tutorial thus far, and if you prepared your homework, you should be in a good shape. You are now a certified *beginner programmer*. Perhaps not quite ready yet to be the Vice President of Engineering at Microsoft, but certainly ready to go into more complex and challenging stuff. All beginnings are hard, and if LogoMation is your first programming language, then you have made a most important step towards mastering this trade.

Congratulations!

Where should you go from here? There are two ways that you can make progress and become a better programmer.

The first is to seek wisdom. No, not necessarily in a monastery in Tibet. Since the invention of the electronic computer, a large number of smart people have used it and devised innovative methods to have it do what they wanted it to do. Your bookstore might have more LogoMation books – check them out. Go to the bookstore and buy a book that will teach you computer algorithms. You would be surprised, and I believe delighted, to find out how many ways there are to sort items on the computer, for example, to sort strings alphabetically. You will discover recursion, and after getting the hang of it, you would love the concept. Before long, you would discover what areas of computer science attract you. It might be computer graphics. Or it might be computations (find out how to compute π to the 1000^{th} digit). If you are a self-learner, there is a tremendous amount of information out there that you might want to access. And if you prefer a more structured study, you would surely find that, too.

The second crucial ingredient in making you a better programmer is practice. The more programs you write, the better you will be at writing the next one. Seems rather obvious, I know. Practice will teach you that programming is an *engineering* activity, rather than a scientific or an artistic activity. The more you code, the more you will understand the engineering tradeoffs that one should constantly consider: should I make this program more general and less efficient, or more efficient and less general? Should I reduce the number of features on this program and release it, or should I spend more time making it more powerful? How should I go about designing this project? How can I maximize the odds that other people would use the program and enjoy it too?
At first, these questions would not bother you much, but as you become more professional, you will find yourself thinking about them more and more. When you find yourself doing it, and when you find yourself inventing ways to objectively measure these tradeoffs, e.g. in time or in money, then you will know that you have become an *experienced* programmer.

Programming can be done alone, and can be done as part of a team. Programming is an extremely rewarding intellectual activity. Oh, and let's not forget that it also has its financial rewards: a programming project can finance your next vacation, and if you are lucky and skilled, your next house.

You are off to a good start. Make yourself a cup of tea, and start reading the Reference Manual part of this book.

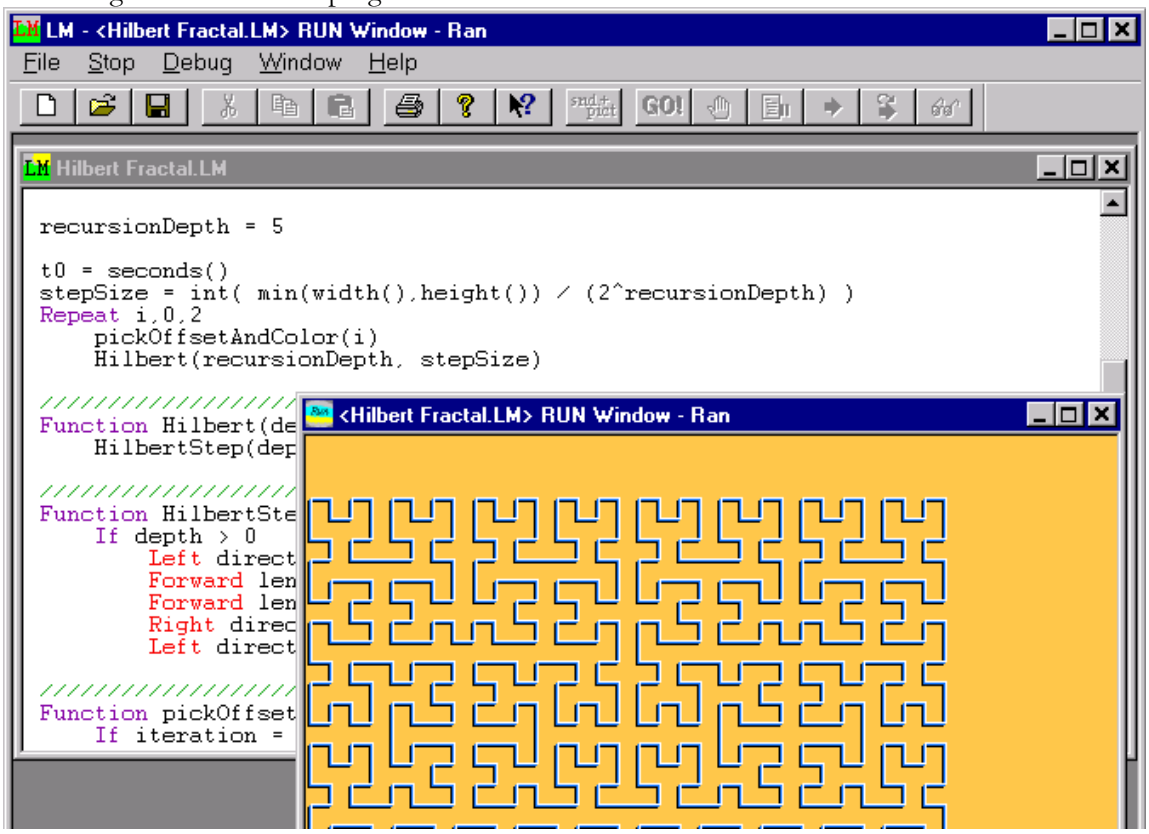
Chapter 1:A Quick Tour

This is the first chapter of the Reference Manual section of this book. If you are reading these lines, then either you have just finished the tutorial, or you chose to skip the tutorial and go directly to Reference Manual section of this book. If you have read the tutorial, this chapter could be taken as a refreshing review, although you surely will learn new things here.

Starting at the next chapter, we will be getting into the gory details. This chapter will give you a glimpse at some of the features of LogoMation. The tour is done in a rather intuitive manner. If you don't understand something, do not worry – it will be explained more rigorously later in the manual.

1.1 The LogoMation Windows

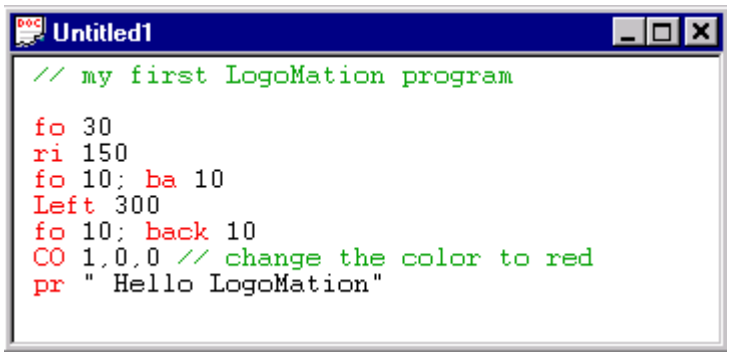
You edit LogoMation programs in Edit windows, and see the results of running them in Run windows. At any moment, any number of Edit and Run windows can be opened. The next screen shot shows a LogoMation application with one Edit window (in the back), and a Run window (front) containing the results of the program's execution.



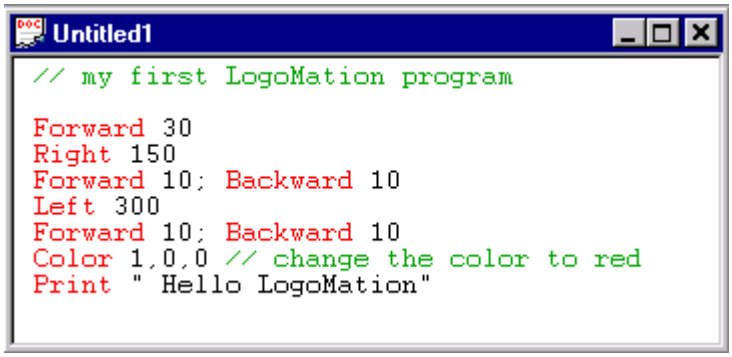
1.2 Hello World – The First LogoMation Program

Using LogoMation, it is very easy to write programs that draw things on the screen. The most basic drawing is done via *pen* movement. Imagine holding a pen and telling it to go places, such as go forward in a straight line for 100 pixels, then turn right 45°, continue 70 pixels in a circle of a given radius, and so on. As the pen moves, it draws a line on the screen, and you can control the line's attributes, such as its width and color.

Let us draw an arrow, and at its tip write "Hello World" in red. To do so, double-click the LogoMation icon to invoke the application. It comes up with an empty Edit window, ready for typing a program in it. Note that as you type, the color of the typed letters indicates how LogoMation understand them. A LogoMation program is made of a series of *statements*. LogoMation statements are written in separate lines, or can be grouped together in one line by separating them with semi-colons. Statements start with a *command* name, which LogoMation colors red or purple (see below). Comments, which are ignored by LogoMation, start with two slashes, end at the end of the line, and are colored green. Here is our first shot at typing the program; you are invited to type it in. Be sure to use **0** (the number zero) when typing a number, and **o** (the letter) in the commands.



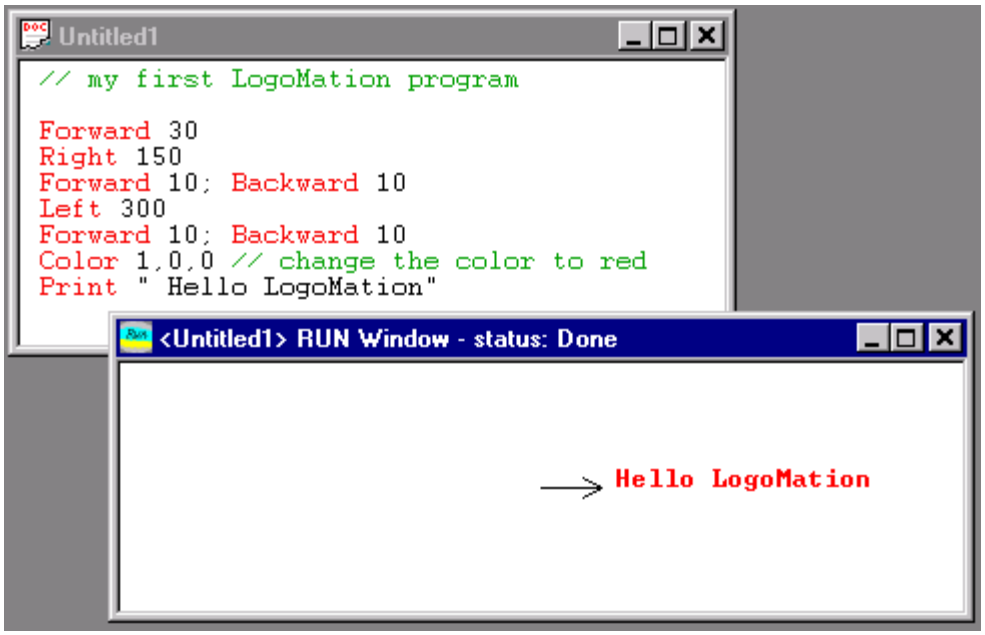
This program is not very legible. It uses cryptic statements like **FO 30**. LogoMation allows an abbreviation of command names, and we have taken advantage of that. The FORWARD command, for example, can be abbreviated and can be typed in any combination of upper and lower case. Thus **FO**, **for** etc., are all legal abbreviations of **FORWARD**. Most users find it easier to type in an abbreviated command, and let LogoMation "beautify" it. This can be done by selecting the lines that need to be "beautified" (or typically, typing ctrl-A to select all the text), and then selecting the Beautify command from the Tools menu. Doing so, we get the following, which is *much* more legible:



If you get any error messages when applying the "beautify" command, click OK in the error box, and correct the highlighted line – you must have typed it erroneously.

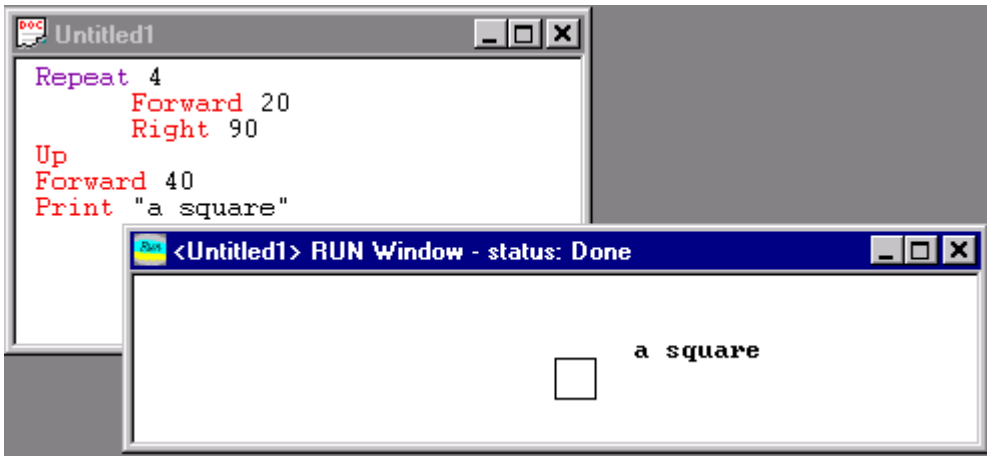
At the beginning of the program, the pen is facing "east" (i.e., to the right). The first statement moves it in this direction for 30 pixels. The second statement changes the direction by 150° clockwise, i.e., it now faces southwest.

We are now ready to click the GO button (alternatively, we could click the **Go** button, or select the **Go** command from the Run menu, or simply type ctrl-G). Here is what we get, after some re-arranging of the windows:



1.3 WYSIWYG Indentation

LogoMation has two types of statements: simple statements, and *compound* statements. The program in the previous section contained only simple statements. Compound statements tell LogoMation that something needs to be done to a *group* of LogoMation statements. For example, the group of statements should be repeated several times, or it should be executed conditionally. In the example below, we use a REPEAT statement to create a square.

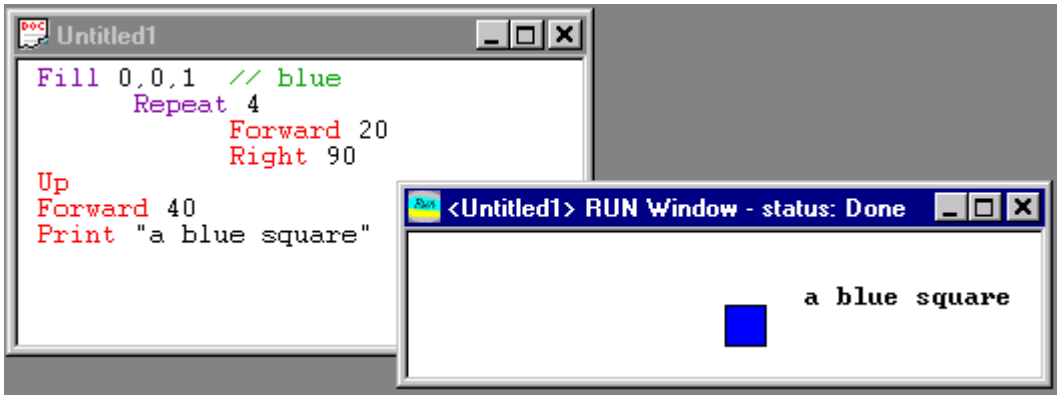


Note that the color of the REPEAT command is purple: LogoMation uses this color to indicate compound statements. The REPEAT statement cause the two lines that follow it to be repeated four times, thus drawing a square. LogoMation knows that only two statements need to be repeated, because these two statements are indented to the right relative to the compound statement. The last three statements in the program are not indented relative to the REPEAT command, and therefore they are not part of the REPEAT statement.

Indentation is aided by LogoMation, but is essentially something that the programmer is responsible for. When you use the "return" (or "enter") key after typing the REPEAT statement, LogoMation

automatically starts a new indentation level, i.e., places the cursor at one indentation step relative to the previous statement. When you type "return" (or "enter") at the end of the following FORWARD statement, LogoMation still maintains the same indentation level. The same is true after the next "return" (or "enter") at the end of the RIGHT statement, except that now we wish to end the indentation so we have to erase one level of indentation. We do so by using the backspace key. In general, you can always indent one level to the right by typing a space at the beginning of a line, indent one level to the left by erasing a space character at the beginning of the line. You can also use the Indent Right and Indent Left menu commands to indent a group of lines.

You might wonder whether compound statements could be nested. The answer is that they do, and in fact it is a very common practice in LogoMation. In the next example, we fill the square with blue:



The FILL statement is used to fill a shape with a given color and pattern. The shape is drawn by the group of statements, which follow the FILL statement and are indented to the right relative to it. In the example above, we fill the square with a solid blue, selected by the three numbers in the FILL statement. The exact way of how to select a color of a pattern will be explained later in this manual.

1.4 The *Tracer*

LogoMation is very fast, but sometimes you may want to slow it down. For example, you could be working on one of these fractals programs that can be expressed in 10 lines of LogoMation code, but might take hours to get right. Each time you click the GO button, and almost instantaneously you get a strange picture which you certainly did not expect.

The *Tracer* is a tool for slowing down the drawing process. It draws lines at the rate of 50 pixels/seconds, so for example a `Forward 100` statement would take 2 seconds to complete. That is enough time to see which line preceded the current line, which direction the line is drawn, and what line follows it. The tracer can be toggled on and off by selecting the **Pen Tracer** command from the **Run** menu. The tracer can also be turned on or off from the program itself, as explained in section 8.4, *Setting a Pen as a* Tracer, on page 107.

> The tracer is only available on Windows. On the Mac, you can instead use "turtle mode", described in section 10.3, *Application – "Turtle Mode"* page 114.

1.5 Program Development and Debugging

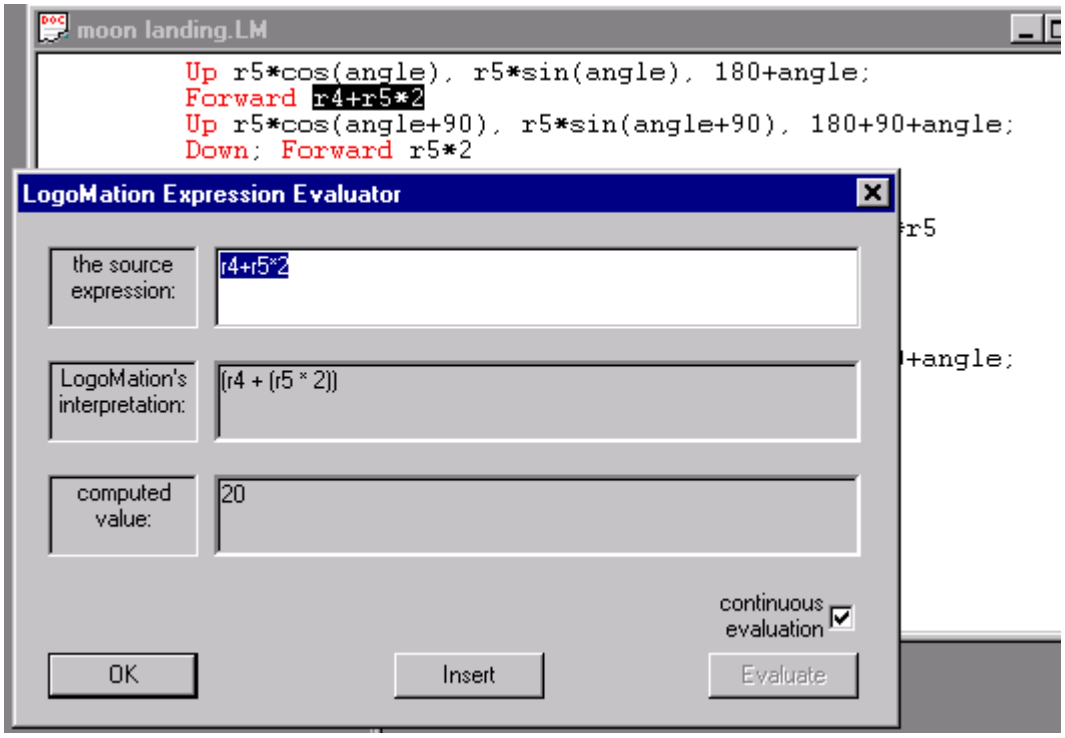
LogoMation uses hundreds of different messages to indicate to the user when something is wrong. This includes compilation errors, run-time errors, and other messages. An example of a compilation error is when a given command cannot be understood. An example of a run-time error is when an

argument of a command has an illegal value. When an error occurs, LogoMation pops up an error message box, and highlights the statement that caused the error.

After a program starts running, it keeps running until one of the following:

1. Normal end – the program's last statement was executed.
2. A HALT statement was executed.
3. Program execution was halted by the user (on Windows: by using the STOP button, or the Stop menu command, or ctrl-Break. On the Mac: via cmd-period).
4. A run-time error had occurred. In that case, LogoMation pops up an error message, and gives the programmer the choice of checking the erroneous line.

After a program execution had been paused or had ended, variables and expressions can be examined by clicking the **Evaluate** button (or selecting the **Evaluate** command from the Run menu). The screen shot below shows an example of evaluating an expression by selecting it in the Edit window and then invoking the evaluator. The user had selected the expression **`r4+r5*2`**. LogoMation shows the selected expression in the first field, the way it understands it (including the order of evaluation) in the second field, and the result of evaluating the expression in the third.



Note that evaluating an expression in this way can only be done after a program's execution has stopped or was paused, and the Run window is still up. You can use the evaluator even if the program has never run, but in that case you will obviously not be able to see the values of variables. Using the evaluator when the program did not run yet is handy for checking the order of expression evaluation (seen in the *LogoMation's Interpretation* field), and also for doing computations that do not include variables, for example, **`2^15-1`**.

1.6 Animation and Sound

The real fun begins when a LogoMation program is used to animate objects on the screen. In essence, animation is done by attaching a picture to a pen, and then moving the picture around, using pen movement statements. The animation can be controlled in several ways, such as setting its speed, or controlling whether it is done under or above the background. Sound can be played, synchronously, asynchronously, and in a loop, in conjunction with the animation.

Pictures can be created in via the PICTURE compound statement, or they can be imported from picture files. Sounds can be recorded in LogoMation (Mac version only), or can be imported from sound files.

We end this quick tour by taking the blue square example one small step forward. First we create a picture called "my blue", and then we move it from (0,0) 100 pixels to the right. Note that when the PICTURE statement is executed, no picture is drawn on the screen. The picture is drawn in the computer's memory and is labeled "my blue". The picture is then attached to the pen using the PEN statement, and is then moved to the right using the familiar FORWARD statement. It's kind of hard to show the animation in a screen shot, so we'd suggest that that you type in the following program and try it out yourself:

1.7 What's Next

Before reading on, you may like to play a bit with LogoMation. Run some sample programs and see if you can guess what certain statements do. You might have fun writing your own programs, even before continue reading this manual, by copying statements from some examples and modifying them. Or you may prefer to read on before attempting to write your own code. Either way, have fun – this is the goal of LogoMation!

