ACSLogo

User Guide

Version 1.4f

Contents

Getting Started 5
- ACSLogo Requirements 5
- Downloading the Program 5
- Tutorials 6

First Steps 7
- Starting the Program 7
- The Main Window 7
- The Graphics Window 7
- Commands 8
- Help for Commands 10
- Other Commands 10
- Command Output 10
- Arithmetic Expressions 11
- Minus Signs 11

The Turtle 13
- The Canvas 13
  - The Turtle's Position 14
- The Turtle's Heading 15
  - Commands affecting Heading 16
  - Visibility 16
- The Pen 17
  - Up or Down? 17
  - Pen Colour 17
  - Pen Width 17

Datatypes and Variables 19
- Numbers 19
  - Operations on Numbers 19
  - Relational operators 20
  - Mathematical Functions 20
- Words 22
  - Operations on Words 22
- Lists 25
  - Operations on Lists 25
- Variables 27

Flow Control 29
- Repeating Commands 29
- Run 31
- Making Decisions 31
- That's It? 31
- Some More Examples 31

Procedures 33
The Procedures Window 33
Parameters 35
Comments 36
Local Variables 36
Outputting Results 36
Recursion 36
While and For 37
Thing 38
Importing Procedures 39
Graphics 40
Colours 40
Transparency and Opacity 41
Drawing Arcs 42
Text 43
Filling Shapes 43
Shadows 45
Paths 46
FillCurrentPath 46
StrokeCurrentPath 46
Saving Paths 47
Text 47
Corners 48
Holes 50
Vector Graphics 55
Exporting Vector Graphics 55
Files 57
File Management Commands 57
File Manipulation Commands 61
Movies 63
Animation in ACSLogo 63
An Example 63
Speech & Music 65
Speech 65
Music 65
Appendix A: Menus 67
The ACSLogo Menu 67
The File Menu 67
The Export Submenu 68
The Edit menu 69
The Special menu 69
The Window Menu 70
The Help Menu 71

Appendix B: Preferences 73
The Turtle Tab 73
The Editing Tab 76
The Localisation Tab 77
Appendix C: Roll Your Own Turtle 78
Appendix D: Localisation 82
Prerequisites 82
The Application Bundle 83
Localisation Tasks 84
GUI Localisation 84
The .nib File 84
Localizable.strings 87
Logo Command Localisation 87
Command Table 90
Help and Tutorial Localisation 93
Tutorials 93
Help 93

Getting Started

ACSLogo Requirements

ACSLogo runs on Mac OSX. The current version (1.4f) runs on Tiger (OSX 10.4) and above. At the time of writing, 'above' is Leopard (OSX 10.5), but it should run OK on the next few versions.

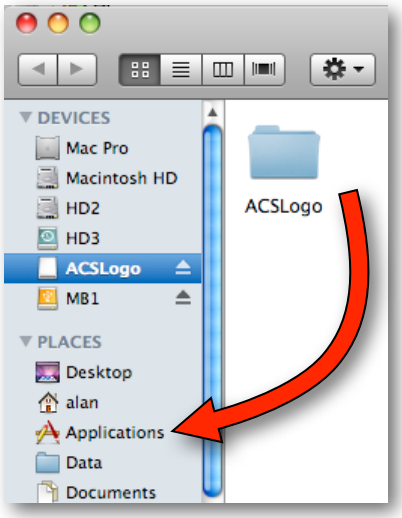
There are older versions of ACSLogo on the website which will work with earlier versions of the Mac OS.

The program will run natively on Intel and PowerPC Macs.

Downloading the Program

Go to www.alancsmith.co.uk/logo and download from the link on the right-hand-side. This downloads a disk image file with the suffix **.dmg** .The Operating System should mount this automatically, but if not, find the file in your downloads folder and double-click it to mount it.

Once it's mounted, drag the enclosed folder to your Applications folder, or anywhere else you want to put it:

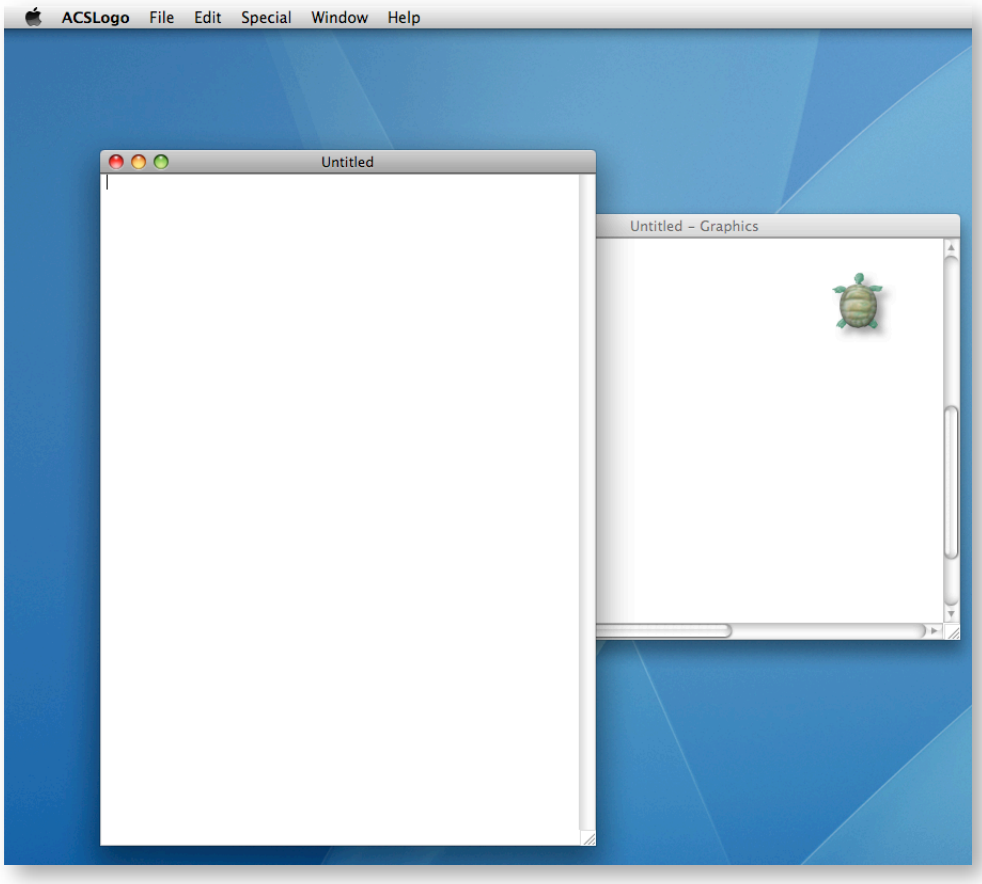


In the ACSLogo folder, you'll find:

ACSLogo.app	The ACSLogo program.
examples.acsl	A file of examples, to show what ACSLogo can do.
Readme.rtf	Some miscellaneous notes about the program.
ReleaseNotes.rtf	What's new in this version.
tutorials	A directory of tutorials.
Website.webloc	Double-click on this to go the website.

(You may not see the file extensions — it depends on your finder settings).

Double-click on the program to start it up. When the program starts, you'll see two windows - a main window called **Untitled**, and another window called **Untitled - Graphics**:



The main window is where you'll type in logo commands. Any drawing done by the turtle shows up in the Graphics window.

You'll probably want to expand the Graphics window to its full size by hitting the green button on its title bar.

Tutorials

Click on the Help Menu, and at the bottom you'll see a list of tutorials from the tutorials folder:

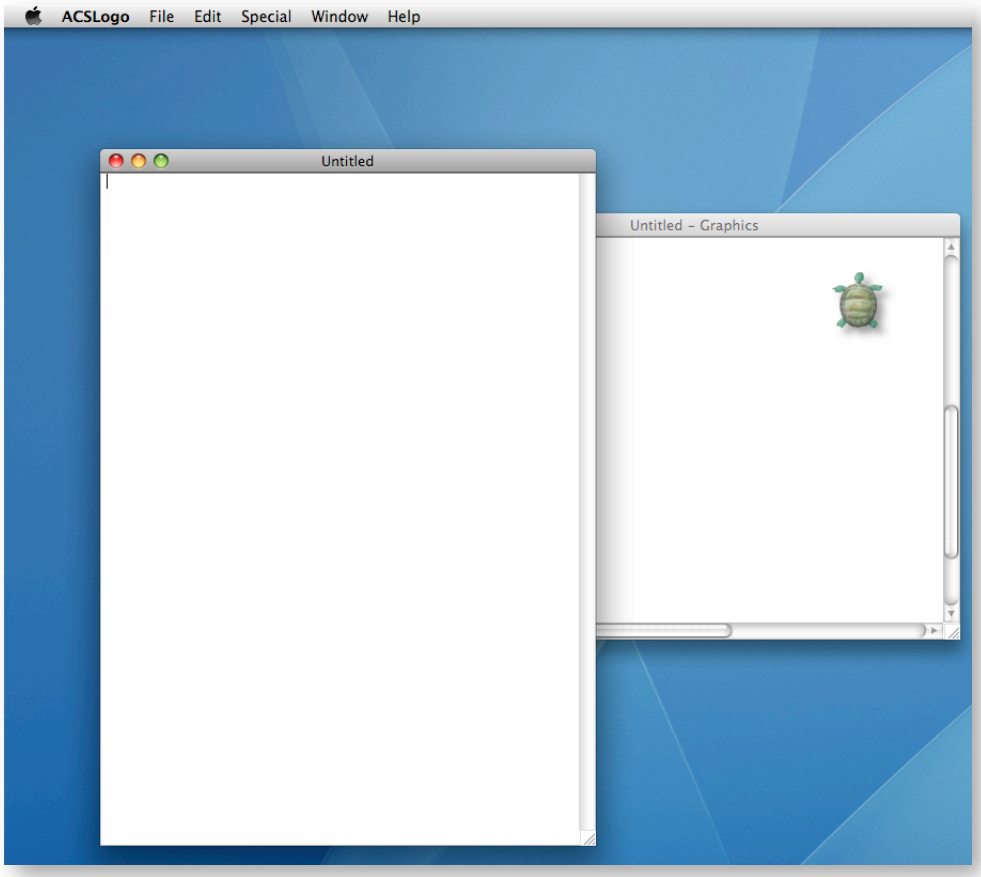
Each of these will open an interactive tutorial about a specific subject. We will cover the same ground in this book, so choose which suits you best.

First Steps

Starting the Program

Start the program by double-clicking on ACSLogo.app (the **.app** suffix may not be displayed depending on your system settings).

When the program opens, you will see two windows – a main window called ‘Untitled’, and a graphics window called ‘Untitled - Graphics’.



The Main Window

This is where you type things in — its main purpose is to type in commands to make the Turtle do some drawing, but you can type in anything you want in the window, change fonts, paste pictures, etc. It's a simple word processor like TextEdit. This is also the place where the program writes out any results or error messages.

The Graphics Window

This is where the Turtle lives, and where it does its drawing. The area that the Turtle draws on is called the Canvas.

We’ll look at some other windows later.

If you can, maximise the Graphics window and position the windows so you can type into the

Main window and see what's happening in the Graphics window.

Commands

In the main window, type this in on a line on its own:

```
Forward 100
```

You should see the Turtle move forward in the Graphics Window, drawing a line as it goes. The turtle has moved forward 100 pixels.

In the Main window, press the Return key to go onto the next line, then type this in:

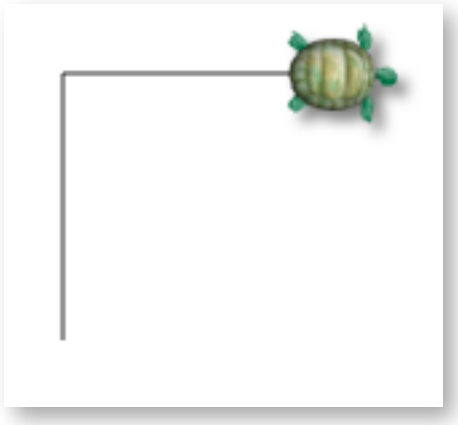
```
Right 90
```

We're going to execute this command, but rather than do it from the menu, hold down the Command key (on the left of the space bar) and press return (you're going to have to execute a lot of commands, and doing it from the keyboard is a lot quicker than going back to the menu each time).

The turtle turns clockwise through 90 degrees.

Now move up to the previous line (the one with **Forward 100**) and press Command-return again.

The turtle moves forward 100 pixels again.



It's obvious from this that when we say **Forward**, where the line is drawn is dependent both on the position of the Turtle and the direction it is facing.

Try using **Forward** and **Right** with different amounts, then try **Left** (which does as you would expect).

If you misspell a command, Logo will complain:

```
Froward 100
unknown function Froward
```

You can press Command-Z (Undo) to get rid of the error message, then go back to correct the command and execute it again. The commands we've looked at so far all take one *parameter*, which is the amount by which to move or turn. If you leave that out, Logo will complain:

```
Forward
wrong number of inputs for Forward
```

Different Logo commands take different number of parameters – one, two, three, or none.

One command that takes none is **ClearScreen**. Execute it now and you will see that it clears the screen and sets the Turtle back to the middle of the canvas, pointing straight up.

Commands are *case-insensitive* — you can type **ClearScreen**, **clearscreen**, or **clearSCREEN**, and they will all do the same thing.

So far, you've typed in the commands individually on separate lines and then executed them one at a time. You can highlight a sequence of commands and press Command-return to execute them together. Type in these commands, highlight them, and execute them together:

```
Forward 100
Right 90
Forward 100
```

You can also type them all in on one line and execute everything on that line:

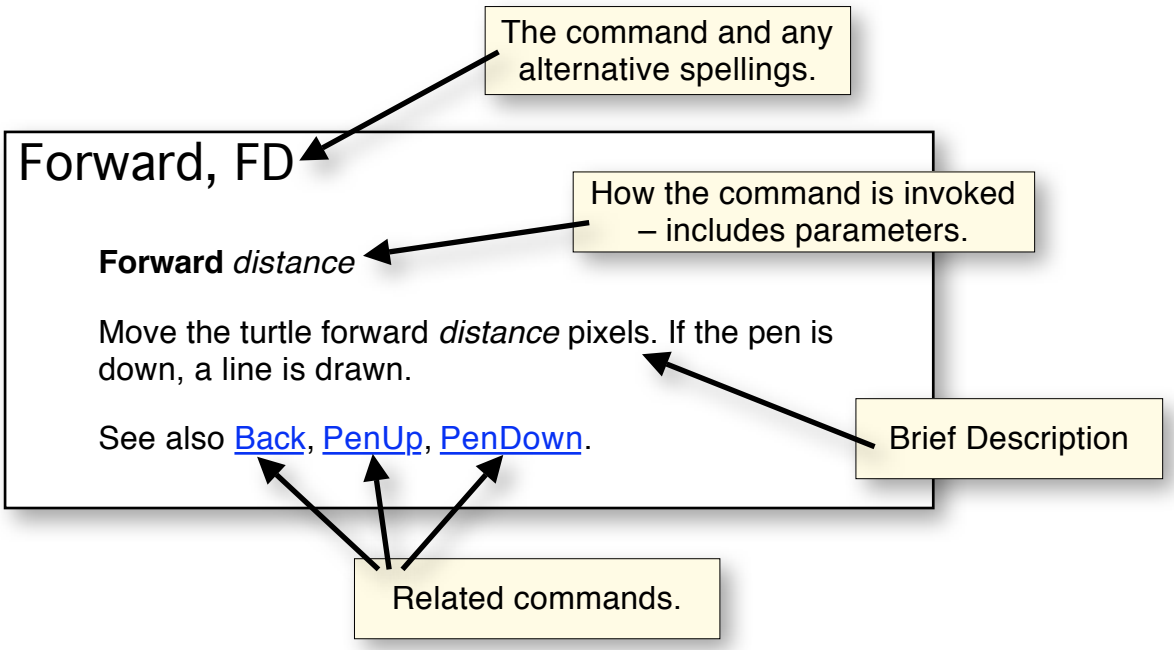
```
Forward 100 Right 90 Forward 100
```

Obviously this isn't quite as clear.

Help for Commands

You can get help for a specific command in two ways — highlight the command in the Main window and then choose **Look Up** from the Help Menu; or hold down the Command key and double-click on the word. Either will bring up the help entry for the command in the Help Viewer window.

Let's look at the entry for **Forward**.



You can see that **Forward** has an alternative, shortened, form – **FD**. The shortened forms of commands can be handy when you're in a hurry.

The next line shows that **Forward** takes one parameter, *distance*.

Next is a description of what **Forward** does.

Finally, there are some links to other commands which are related in some way.

For many commands, you'll also see some examples of their use.

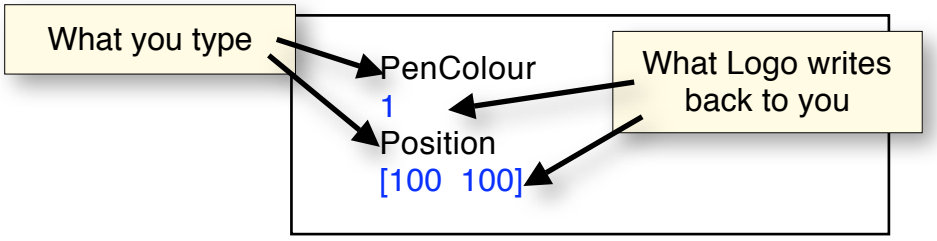
This information can also be found in the *ACSLogo Command Reference* on the ACSLogo website.

Other Commands

We've only looked at **Forward**, **Left** and **Right** so far – these are the commands you will use most of all, but there are many more commands — graphic commands like **SetPenColour** and **SetPenWidth** to change the colour and width of the line drawn by the Turtle; commands which are mathematical functions such as **cos**, **sin**, **tan**; control commands such as **Repeat** and **If**. We'll come across these commands and many others in the following chapters.

Command Output

Many commands write information back to you:



Arithmetic Expressions

Anywhere you can write a number, you can write an arithmetic expression:

```
Forward 100 - 10
SetPenColour 2 * 3
```

Logo evaluates the expression before passing it to the command.

You can also just type in an arithmetic expression, execute it, and Logo will evaluate it for you:

```
15 * 22.6 / 17
19.9412
```

As a lot of commands output values, you can pass the output to another command. Here, **PenColour** returns the number of the current pen colour, and **SetPenColour** sets it to a new value, one greater than the old value:

```
PenColour
1
SetPenColour PenColour + 1
PenColour
2
```

Minus Signs

You might think that a minus sign is just a minus sign, but actually there are two flavours:

Unary minus is part of the number. Binary minus is an operator which has two numbers as its arguments.

It's important that Logo can tell the difference easily. Imagine we've created a procedure called

Proc1 which takes two numeric parameters. What are the parameters in this call?

```
Proc1 5 -4 -3
```

Is **5** the first parameter, or the result of **5 - 4**? What is the second parameter? **-3** or the result of **-4 - 3**. Or have I mistakenly specified three parameters?

For these reasons, unary minus has to be immediately followed by a number with no intervening space, and binary minus has to have a space between it and the following number.

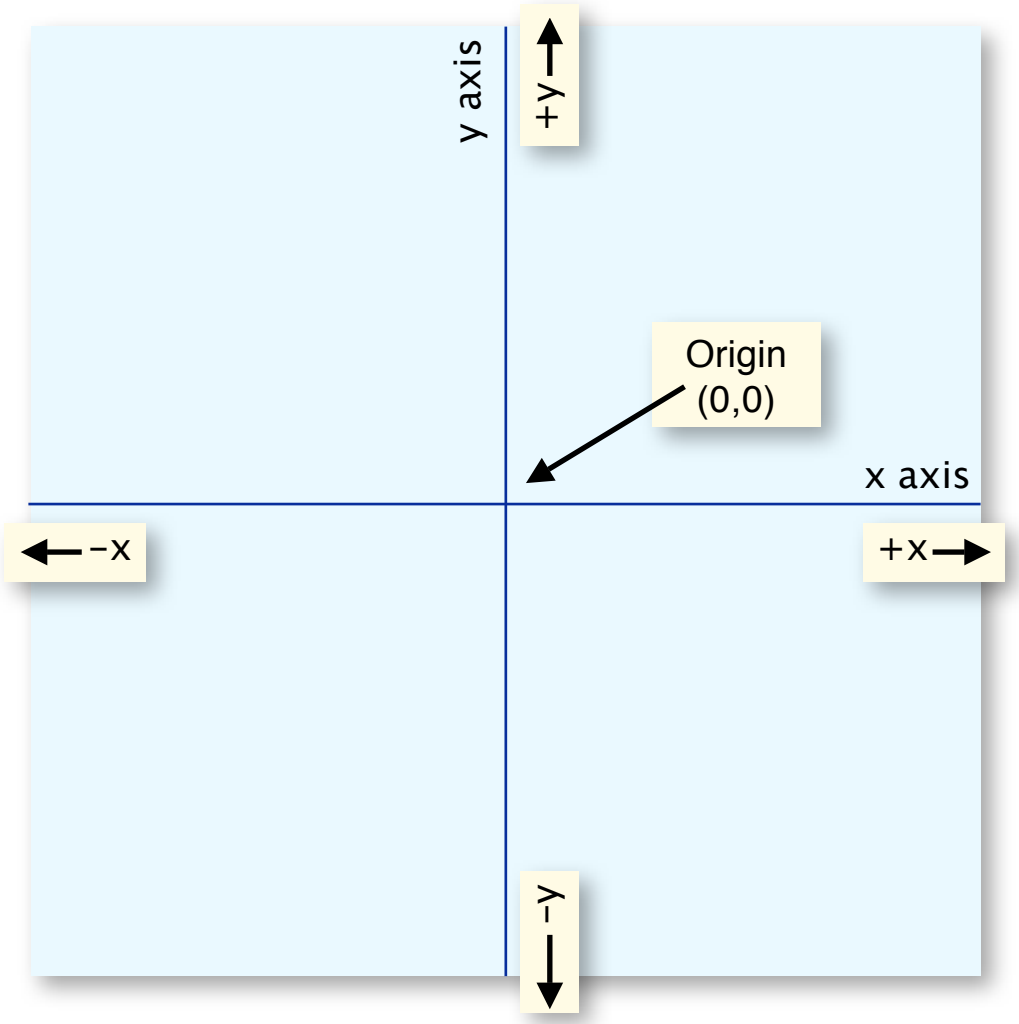
The Turtle

Before diving deeper into graphics, we need to understand the Turtle and how its status affects drawing in the Graphics window. The most important thing about its status is its location, and to understand that we need to look at the Canvas.

The Canvas

The Canvas is the Graphics window with all the gubbins — scroll bars, title bar, etc — taken away. It's where the turtle draws.

Imagine the canvas as a piece of graph paper.



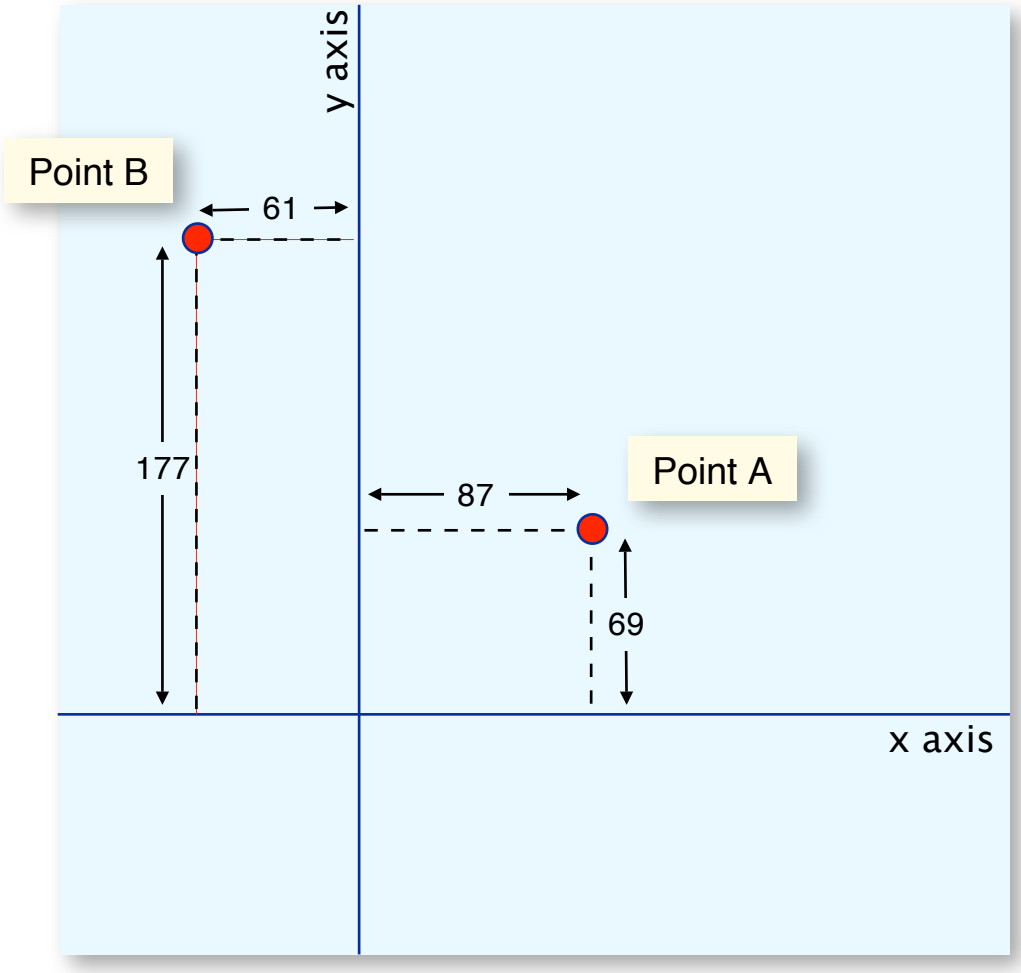
Half-way down the canvas is an imaginary horizontal line going from left to right — the *x axis*. This is used to measure the distance across the canvas. The position half-way along the x axis has an x-value of zero. Positions to the right have progressively larger values; positions to the left have progressively smaller values.

Half-way across the canvas is an imaginary vertical line going from bottom to top — the *y axis*. This is used to measure the distance up and down the canvas. The position half-way up the y axis

has a y-value of zero. Positions above this have progressively larger values; positions below it have progressively smaller values.

The axes (plural of *axis*) meet in the middle, where they both have a value of zero. This is called the origin. Any point on the canvas can be specified by stating its x and y values in the form (*x*,*y*). The x and y values are know as the point's co-ordinates. So the origin has co-ordinates of (0,0).

Let's zoom in a bit closer and look at a couple of arbitrary points on the canvas. We'll call them *Point A* and *Point B*, shown by red circles in the diagram:



Remember, where the x- and y-axis cross, the values of x and y are zero. All along the y axis, the value of x is zero, and all along the x axis, the value of y is zero. Point A is 87 pixels to the right of the y axis, so its x co-ordinate is 87. It is 69 pixels up from the x axis, so its y co-ordinate is 69. Point A is therefore at (87,69).

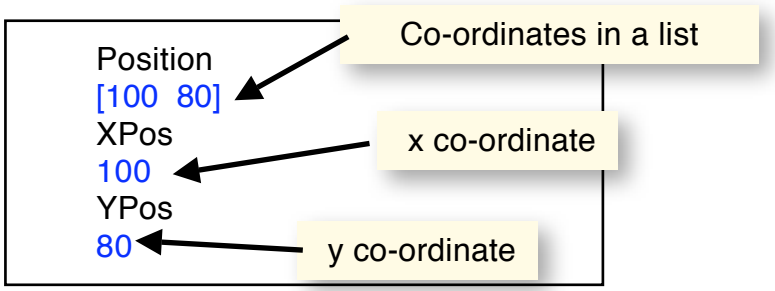
Point B is 61 pixels to the left of the y axis, so it has an x co-ordinate of -61. It is 177 pixels above the x axis, so it has a y co-ordinate of 177. Point B is therefore at co-ordinates (-61,177).

You can change the size of the Canvas (and therefore the Graphics window) using menu **Special/Canvas Size…** — you'll then get prompted for the new width and height; or with the command **SetCanvasSize**. For instance **SetCanvasSize [600 480]** will give you a canvas which is 600 pixels wide and 480 pixels high.

The Turtle's Position

When you start ACSLogo, open or create a document, or issue the command **ClearScreen**, the Turtle is positioned at the origin, co-ordinates (0,0). It is actually the centre of the Turtle that's at

this point. As the Turtle moves around the Canvas in response to your commands, its position changes, but it always knows where it is, and you can query its current position by issuing some commands:



The Turtle's position changes when you issue Forward or Back commands, but you can set the Turtle's co-ordinates explicitly with these commands:

SetPosition [*x-co-ordinate y-co-ordinate*]

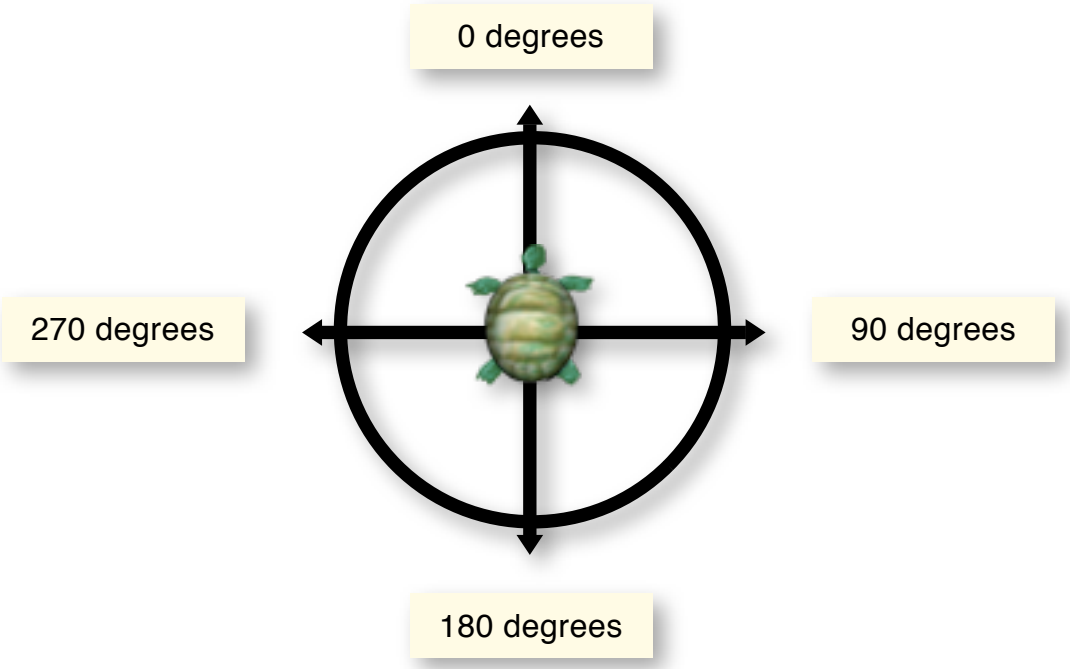
SetX *x-co-ordinate*

SetY *y-co-ordinate*

The **Home** command, like **Clearscreen**, moves the Turtle to the origin, but without clearing the screen.

The Turtle's Heading

The Turtle's heading is the direction it's facing. It's measured in degrees, so as there are 360 degrees in a circle, the Turtle's heading can vary from 0 to (just under) 360. A heading of zero is straight up.

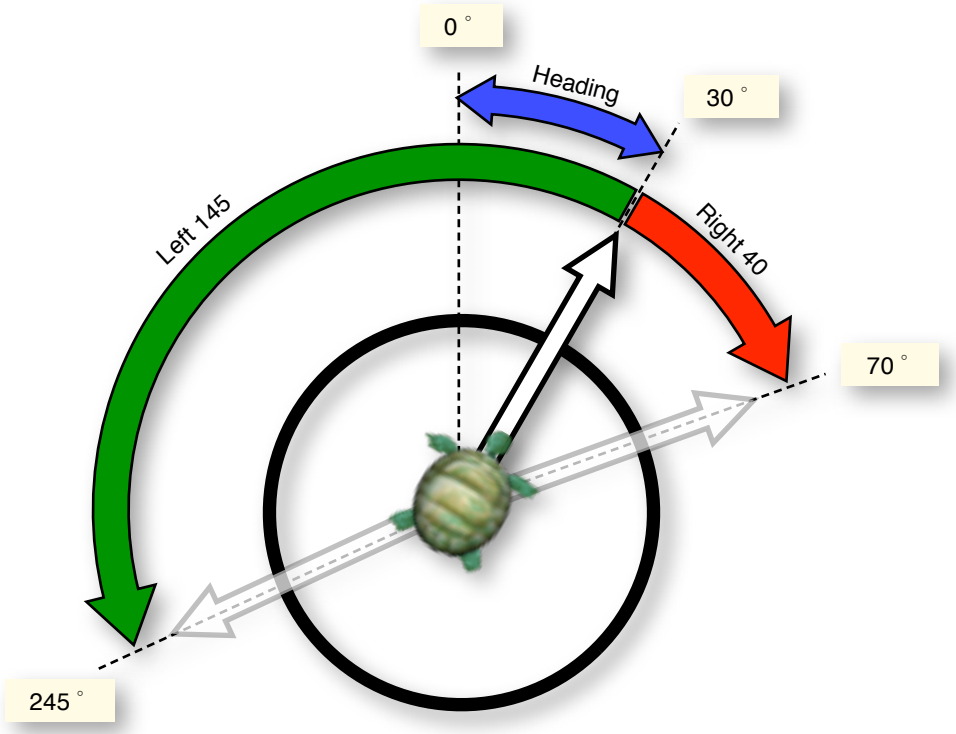


When the Turtle turns right (clockwise), the heading increases; when it turns left (anti-clockwise), the heading decreases. This doesn't look quite right in the diagram above, because turning left 90 degrees from a heading of zero gives a heading of 270. If you subtract 90 from zero,

you get -90 — but we always specify the heading as being between zero and not-quite 360, so if you have a negative heading, just keep adding 360 to it until it's zero or above.

Similarly, going right 90 degrees from a heading of 270 degrees you might expect to give a heading of 360 degrees — but again, the heading is always specified as being less than 360 degrees, so if you get a value of 360 or above, just keep subtracting 360 until you get less than 360.

The following diagram shows a turtle tilted 30 degrees to the right from the vertical (blue arrow), so it has a heading of 30 degrees.



If it turned another 40 degrees to the right (red arrow), it would have a heading of 30 + 40 = 70.

If instead it turned 145 degrees to the left (green arrow), it would have a heading of 30 - 145 = -115. But remember we always give the heading as between zero and just less than 360, so the heading is -115 + 360 = 245.

Commands affecting Heading

As we've seen, **Right** and **Left** affect the Turtle's heading, adding or subtracting from it respectively.

You can also set the heading explicitly using **SetHeading** which takes a single numeric parameter.

In addition, **ClearScreen** and **Home** set the heading to zero.

Visibility

The Turtle can be hidden with **HideTurtle** — This can make it easier to see what you've drawn. It can be shown again with **ShowTurtle**. **Shown?** lets you query the Turtle's visibility.

The Pen

What does the Turtle draw with? — A pen of course! The concept of a pen is used to hold the attributes that affect the lines drawn by the Turtle. The Pen is considered to be right in the middle of the Turtle.

Up or Down?

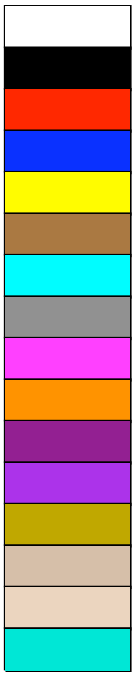
When we move the Turtle, we usually want it to draw a line, but sometimes we just want to move it to a new position without doing any drawing. To do that, we 'lift the Pen off the paper' so that it doesn't draw anything. The command to do that is PenUp. After a **Penup**, no lines will be drawn until you issue a **PenDown**.

Pen Colour

Drawing everything in black would be pretty dull, so we can change the Pen's colour — the colour it draws lines with — using **SetPenColour**, which takes a single numeric parameter. So what does **SetPenColour 3** mean? What colour is 3? Here, the 3 is not a colour in itself, but an index into a list of colours.

This is the list of sixteen colours when ACSLogo starts up. The first colour is Colour 0 (zero) at the top. The last in this list is Colour 15. Beyond that, the colour numbers just repeat the first 16, so colours 16 – 31 are the same as colours 0 – 15, etc. However, you can set any colour number to any colour you like, so this should not be restrictive. See the graphics chapter for how you can change colours with **SetRGB**.

So, to go back to our example, **SetPenColour 3** will set the Pen's colour to the fourth item in the list, which is blue.

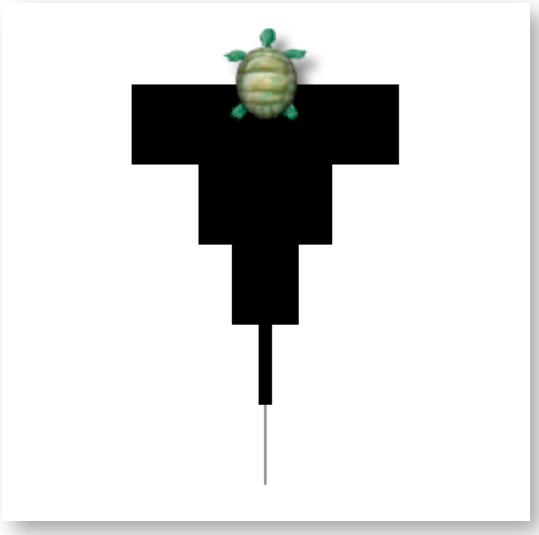


PenColour returns the current Pen colour. **Pen** returns a list containing the Pen's state (**PenUp** or **PenDown**) and the Pen colour, and **SetPen** sets these attributes from a list.

Pen Width

You can change the width of the line the pen draws to get thick or thin lines. SetPenWidth can be used to change the width of the pen from less than a pixel to bigger than the canvas.

```
SetPenWidth 1 Forward 60
SetPenWidth 10 Forward 60
SetPenWidth 50 Forward 60
SetPenWidth 100 Forward 60
SetPenWidth 200 Forward 60
```



PenWidth returns the current width of the Pen.

Datatypes and Variables

We've already seen the use of numbers in Logo — usually we pass them to a command such as Forward. In this section we'll look a bit more at numbers and what we can do with them; then we'll look at the other datatypes that Logo can handle — words and lists. Finally we'll look at variables which can hold the values of numbers, words, and lists.

Numbers

There are two types of number in Logo — whole numbers (integers) and floating-point numbers (real numbers, numbers with a decimal point). Logo can convert between them, so you don't need to worry about what sort of number you've got.

Something to watch out for — if you've got a number between 0 and 1, let's say 0.7 — you have to keep the leading zero: Logo won't accept .7 as a number.

Operations on Numbers

You can use the usual arithmetic operators:

+	multiplication
-	subtraction
/	division
*****	multiplication

You can just enter an arithmetic expression in Logo and the program will evaluate it (output in blue):

```
5 + 6
11
12.3 * 100
1230
10 / 3
3.33333
6 - 4
2
5 + 6 * 3
23
(5 + 6) * 3
33
```

A couple of things to note here – Logo follows the usual rules for operator precedence: multiplication and division have a higher precedence than addition and subtraction. This means that ***** and **/** are evaluated before **+** and **-**, so in the fifth example above, **6** is multiplied by **3** first, then **5** is added to the result. If you want the addition to happen first, you need to use parentheses

(as in example 6).

The other thing to note is that when the minus sign is used in subtraction, as in example four above, there needs to be a space between it and the second operand. Otherwise this would look like a negative number, **-4**.

Relational operators

<	less than
>	greater than
=	equals

These are used in comparisons. They return the value **true** or **false** depending on the comparison:

```
5 < 6
true
5 > 6
false
3 = 3
true
4 = 3
false
```

Using the **Not** function reverses the boolean result:

```
Not 5 < 6
false
Not 5 > 6
true
Not 3 = 3
false
Not 4 = 3
true
```

We will see the importance of the relational operators in the chapter Command and Control.

Mathematical Functions

There are a whole load of trigonometric functions: **Cosine**, **Sine**, **Tangent**, **ArcCosine**, **ArcSine**, **ArcTangent**.

Here are quick descriptions of some of the other number functions:

Abs	Output the absolute value of a number
Exp	Returns e to the power of the input number
Integer	Truncates the input number to its whole number portion
Log	Returns the natural logarithm of the input number
Log10	Returns the base-10 logarithm of the input number
Pi	Outputs the value of Pi
Power	Returns parameter1 to the power of parameter2
Random	Returns a random integer between zero and the input number
Remainder	The remainder when parameter1 is divided by parameter2
Round	Round the input to the nearest integer
Sqrt	Output the square root of the input number

Some examples:

```
Abs -11
11
Integer 1.8
1
Round 1.8
2
Pi
3.14159
Random 10000
432
Power 10 3
1000
Log10 1000
3
Sqrt 225
15
```

Words

A word is just a sequence of characters. All these are words:

```
Herbert
schizoid
xyz
too much!!!
456
```

When you specify these to Logo, you must precede the word by a double quote character("). This signifies to Logo that the sequence of characters is a word, rather than the name of something like a function or variable.

In other programming languages, these are known as strings, and I will sometimes refer to them as such.

When you specify a word, Logo needs to know where it ends. When it comes across a space, or end of line, or an operator such as a plus sign, it terminates the string there. The trouble is that sometimes we want the word to contain a space or other delimiting character. We can get round the problem by using an escape character, the backslash (\).

Here are some examples:

```
"QWERTY
QWERTY
"Too\ hot
"Too hot
"A\ backslash\ looks\ like\ this\ \-\ \\
"A backslash looks like this - \
```

We'll see later that it's easier to use lists to get around the problem.

Operations on Words

The arithmetic operators don't work with words, but the relational operators do:

```
"abc = "abc
true
"abc < "xyz
true
"abc < "ABC
false
```

There are a whole lot of functions for operating on words.

First we have two functions for telling you about the size of a word.

Count	Returns the number of characters in the word
Empty?	Returns true if the word has no characters

```
Count "QWERTY
6
Count "Too\ hot
7
Empty? "abc
false
Empty? "
true
```

Then we have functions for accessing parts of a word.

First	Returns the first character of the word
Last	Returns the last character of the word
ButFirst	Returns all except the first character of the word
ButLast	Returns all except the last character of the word
Item	Returns a single character of the word

```
First "QWERTY
Q
ButFirst "QWERTY
WERTY
Last "QWERTY
Y
ButLast "QWERTY
QWERT
Item 3 "QWERTY
E
```

Then we have functions for constructing new words from other words:

FirstPut	Returns the first input in front of the second input
LastPut	Returns the first attached to the end of the second
Word	Outputs the input words concatenated

```
FirstPut "ABC "XYZ
ABCXYZ
LastPut "ABC "XYZ
XYZABC
Word "ABC "XYZ
ABCXYZ
```

So what's the difference between FirstPut and Word?

Word can take several input parameters, not just two as shown above. To do this, you have to enclose the whole function call in parentheses. For example:

```
(Word "ABC "DEF "GHI "JKL "MNO)
ABCDEFGHIJKLMNO
(Word "Too "Much "Too "Young)
TooMuchTooYoung
```

Just a few miscellaneous functions left:

Word?	Output true if the parameter is a word
ASCII	Output the ASCII code of the first character of the input.
Char	The opposite of ASCII - outputs the character for an ASCII code
LowerCase	Outputs a word with all uppercase characters as lowercase
UpperCase	Opposite of LowerCase
Member?	Output true if the first parameter is a member of the second

```
Word? "ABC
true
ASCII "a
97
Char 97
a
LowerCase "ABC1234\ abZ
abc1234 abz
UpperCase "ABC1234\ abZ
ABC1234 ABZ
Member? "a "ABC1234\ abZ
true
```

Note that for **Member?**, this only works when the first parameter is a single character.

Lists

A list is like a word in that it is a sequence of objects, but whereas a word consists of characters, a list can contain a number of different objects – words, numbers, or even other lists.

When you represent a list in Logo, you surround it with square brackets.

All of these are lists:

[A B C]

[1 2 3 4]

[abc def ghi jkl]

[1 27 [56 45] [12] [[v] g]]

[ ]

The last example is the empty list – one having no members.

Note that words within the list don't need to be preceded by a double quote (").

Spaces within the list separate its elements.

When you specify a list to Logo, characters between the [and] are read unchanged, so there's no need for the escape character. This makes it much easier to get a long sentence into a list than into a word. So this is a valid list:

[c d * / "]

Operations on Lists

The only relational operator you can use is the **=** operator (less than and greater than don't make much sense for lists). Try these examples:

```
[A B C] = [ A  B  C ]
true
[a b c] = [[a] [b] [c]]
false
Not [a] = []
true
```

A lot of the functions we saw used with words also work with lists:

Count	Returns the number of characters in the list
Empty?	Returns true if the list has no members
First	Returns the first member of the list
Last	Returns the last member of the list
ButFirst	Returns all except the first member of the list
ButLast	Returns all except the last member of the list
Item	Returns the nth member of the list

FirstPut	Makes the first input the first member of the second input
LastPut	Makes the first input the last member of the second input

```
Count [A XYZ [a b c]]
3
Empty? [ ]
true
First [ABC DEF GHI]
ABC
Last [47 45 [ ]]
[ ]
ButFirst [Ax 23 [g h i] 29]
[23 [g h i] 29]
ButLast [Ax 23 [g h i] 29]
[Ax 23 [g h i] ]
Item 3 [zzz [f g h] 27 3.4]
27
FirstPut "the [end]
[the end]
LastPut "stop [full]
[full stop]
Member? "abc [def abc ghi]
true
```

Instead of the function **Word** used to construct words, we have a couple of functions to help construct lists, **List** and **Sentence**. These examples show the similarities and differences:

```
List 123 789
[123 789 ]
(List 57 48 123 110)
[57 48 123 110 ]
(Sentence 55 66 77)
[55 66 77]
List [abc] [def]
[[abc] [def] ]
Sentence [abc] [def]
[abc def]
```

As you can see, Sentence strips off the outer brackets of the objects before it puts them into the list.

Just one more list function, **List?**. This outputs **true** if its parameter is a list:

```
List? [New York London Paris Munich]
true
```

Variables

Up to now, all the values we've been inputting to functions have been literal values or literals.

Like all other programming languages, Logo can hold a value in a variable. A variable can hold any type of value – a number, word, or list, or even a boolean (true or false) value.

The two things we can do with a variable are set its value (assignment) and get its value.

You need to give the variable a name – don't have spaces or any other odd characters in the name and don't start it with a number.

To create a variable myNumber and set its value to one:

Make "myNumber 1

Then to get its value back:

:myNumber

Note that when setting the variable, we use **"** , when getting its value, we use **:**.

To increase the value of myNumber by one:

Make "myNumber :myNumber + 1

I can pass the value of myNumber to a function as if it was a literal number:

SqRt :myNumber

I can, if I like, then set the variable myNumber to contain a different type of object, such as a list or a word:

Make "myNumber [abc def]

Make "myNumber "thingie

So a variable can be used to pass a value to a function. There is though, a situation where that doesn't work. Remember the function **SetPosition** which sets the position of the Turtle from its parameter, a list of two numbers. We can try and pass variables to it:

```
make "x 20
make "y 50
SetPosition [:x :y]
SetPosition needs a list of two numbers
```

Remember that when you put things in a list, they appear in the list in exactly the same form as you wrote them, so the first member of the list is not **20**, but the word **:x** . This is easily demonstrated:

```
[:x :y]
[:x :y]
```

The list has to be constructed dynamically using the List command:

```
List :x :y
[20 50 ]
```

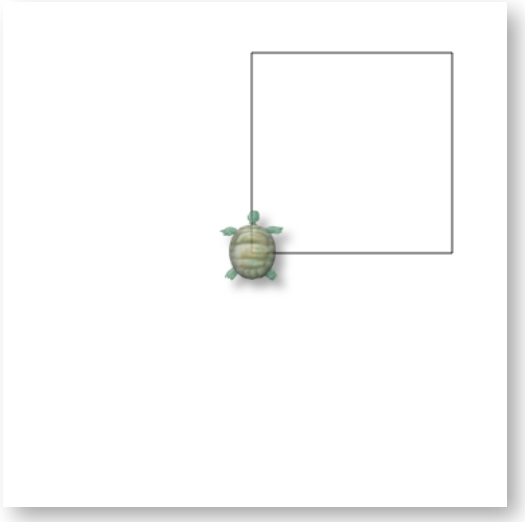
So this can be passed to **SetPosition**:

```
List :x :y
[20 50 ]
```

Flow Control

So far, we've been issuing single commands, or a number of commands one after the other in a block, like this sequence of commands to draw a square (Fd is short for **Forward** and Rt is short for **Right**):

Fd 150 Rt 90 Fd 150 Rt 90 Fd 150 Rt 90 Fd 150 Rt 90



In this tutorial we'll be looking at some commands which will make it easier to issue repeated sets of commands and a command for deciding whether to issue a command.

Repeating Commands

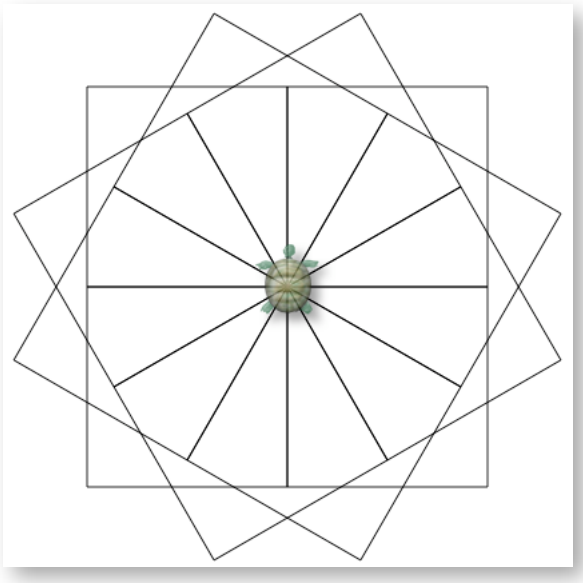
The example above has two commands (Fd 150 Rt 90) repeated four times. Rather than have to write them out four times, we can use this construct:

Repeat 4 [Fd 150 Rt 90]

Repeat takes two parameters — a number and a list. The number specifies how many times the commands in the list are to be executed.

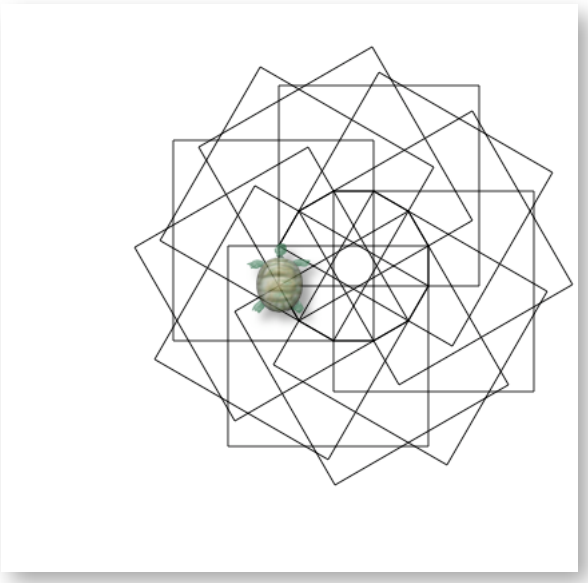
We can build up complex patterns using Repeat. Here we've got an inner Repeat drawing the box as before, while an outer Repeat draws the box 12 times, rotating through 30 degrees each time:

CS Repeat 12 [Repeat 4 [Fd 150 Rt 90] Rt 30]



and here's a variation on the last pattern:

CS Repeat 12 [Repeat 4 [Fd 150 Rt 90] Fd 30 Rt 30]



Often when you're looping like this, you want to keep a count of what iteration of the loop you're on. You can use a variable to do this. Say you want to print the first ten numbers. First you initialise your variable:

Make "myNumber 1

Then Repeat the commands which will print the value of the variable and increment the variable:

```
Repeat 5 [Print :myNumber Make "myNumber :myNumber + 1]
1
2
3
4
5
```

Run

For the case where you just want to run the commands in the list once, there is the **Run** command. For example:

Run [ClearScreen Fd 200]

This looks like a fairly useless command — why not just run the commands in the list?

The power of this is that you can build a list of commands, store them in a variable and then run the list later:

Make "cmd [ClearScreen SetPenWidth 20 Fd 200]

Run :cmd

Making Decisions

Logo has the **IF** statement to decide what action to take depending on whether a condition is satisfied. An **IF** statement looks like this:

IF condition [*true statements*] [*false statements*]

The condition is an expression which returns true or false. We've seen examples of these sort of statements already:

```
5 > 6
false
Empty? [ ]
true
"the\end = Word "the "end
true
```

Of course, normally you'll be comparing the values of variables rather than literals to literals.

If the condition is **true**, the first list is executed, otherwise the second list is executed. Here's an example:

```
Make "amount 20
if :amount < 30 [print "small][print "big]
small
```

That's It?

We seem to be missing a few constructs that are available in other programming languages: **while**, **for**, **repeat until**. There's no equivalent in the Logo language, but you can get the same effects using procedures - see the Using Procedures chapter.

Some More Examples

Here's just a few more examples using **Repeat** for you to try.

This example creates a spiral by reducing the amount to turn right each time through the loop:

Make "amount 20

CS Repeat 180 [Fd 8 Right :amount Make "amount :amount * 0.99]

Or something more angular:

```
Make "amount 10
CS Repeat 180 [Fd :amount Right 90 Make "amount :amount  + 8]
```

I can multiply amount by -1, making its sign alternate between a negative and positive value. Going right by a negative amount means going left:

```
Make "amount 4
CS Repeat 40 [Repeat 40 [Fd 1 Right :amount ]Make "amount :amount * -1]
```

Procedures

Procedures are a means of making your logo code easier to understand by grouping a series of commands together. It's a bit like creating your own commands.

Let's consider drawing a square. To create one with a side of 300 pixels, we could say:

repeat 4 [forward 300 right 90]

or to use abbreviations:

repeat 4 [fd 300 rt 90]

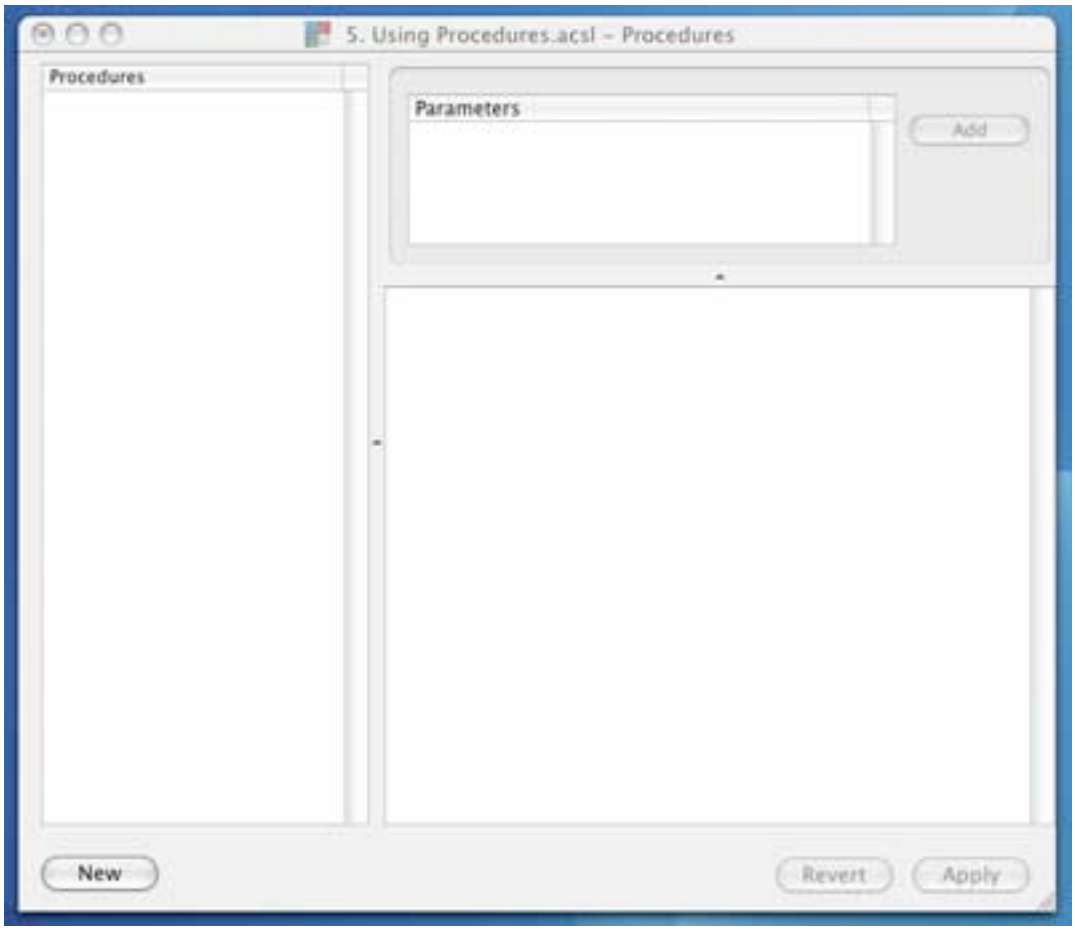
If I then want to make a pattern from the squares by rotating in a circle, I get:

repeat 36 [repeat 4 [fd 300 rt 90] rt 10]

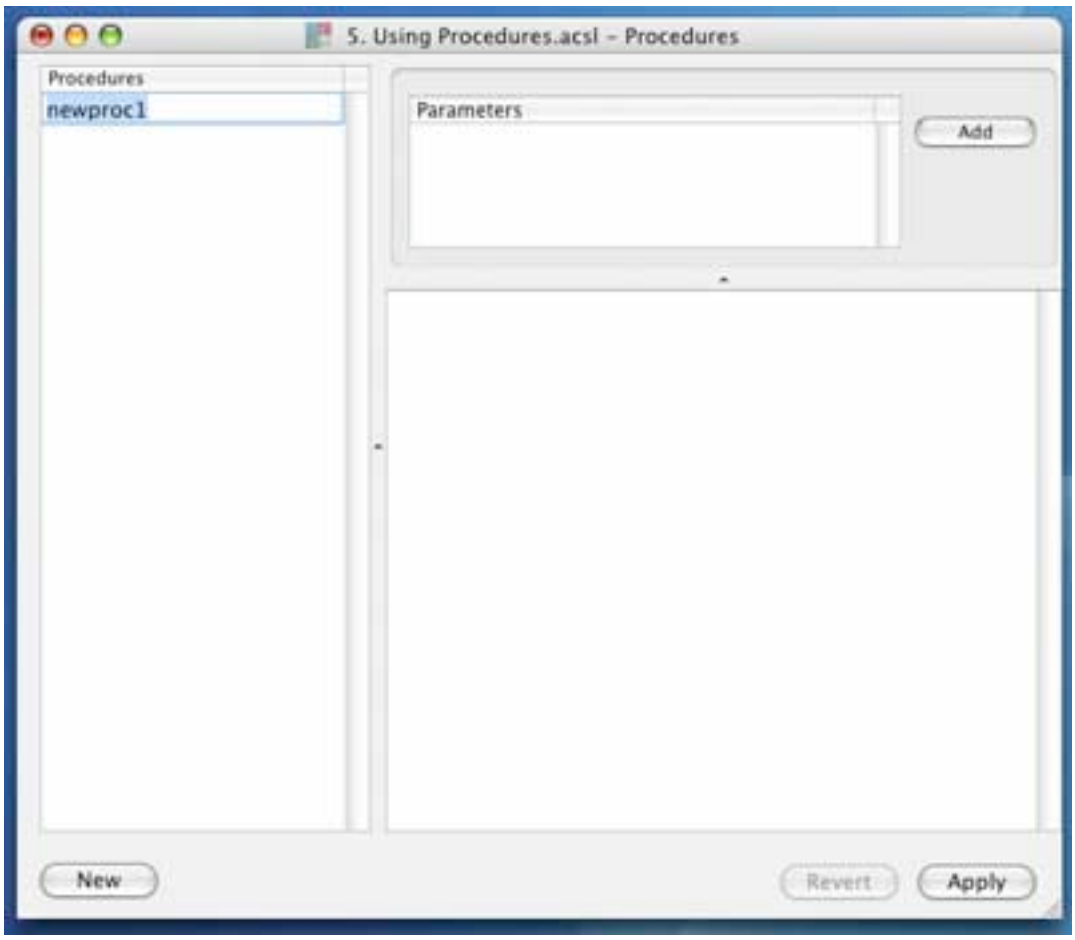
Things soon start to get difficult to understand. To make it simpler, I'll make my commands to draw a square into a procedure called **box**, so whenever I call **box**, I'll get a square drawn.

The Procedures Window

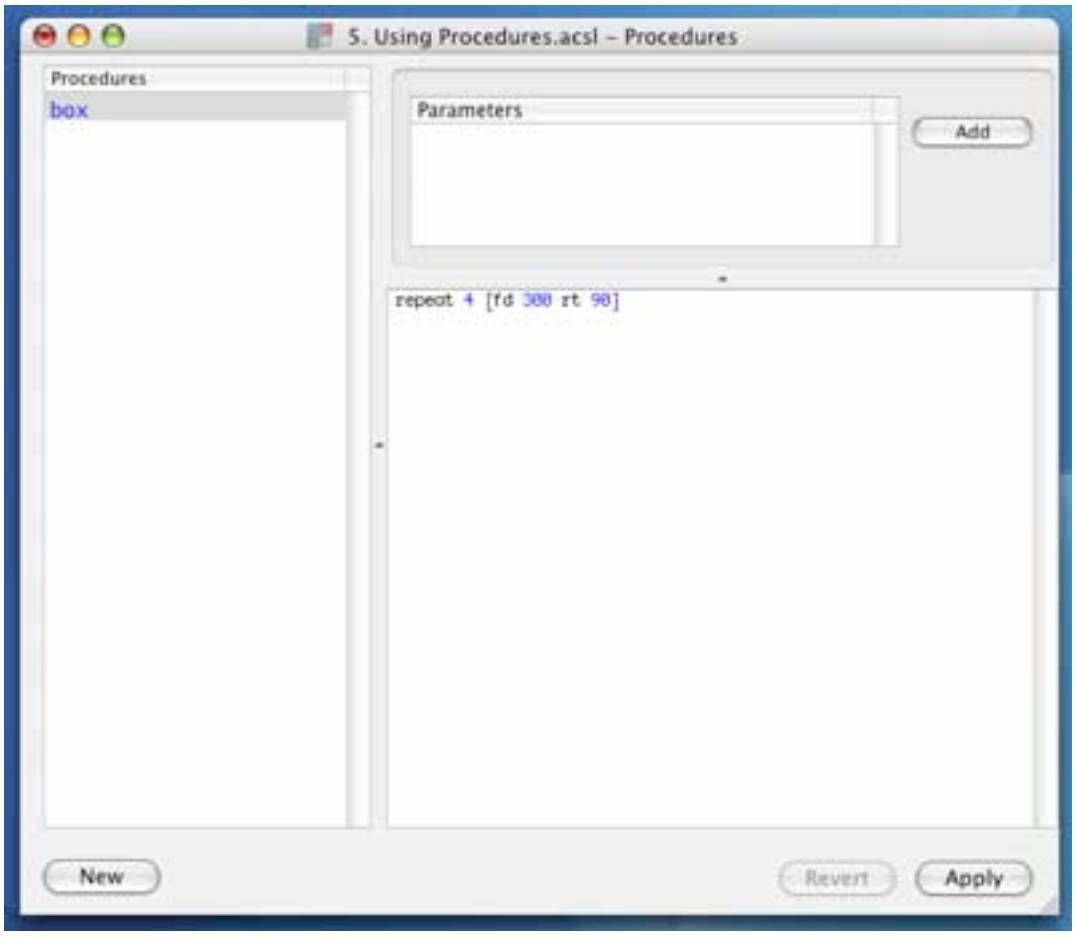
Select menu Window/Procedures to open the Procedures window:



On the left is a list of the procedures you've got defined. Currently there is none. We're going to create a new one, so click on the **New** button:



An entry is added to the list and assigned the name **newproc1**. Overtype this with the name **box** and press return. Type the box-drawing commands in the main text area:



Note that the name of the procedure is blue because the procedure has been changed (created actually) but has not been applied. Press the Apply button to make the changes available.

We can now call the box procedure (after clearing the screen). Issue this command:

cs box

to invoke the commands in the procedure box.

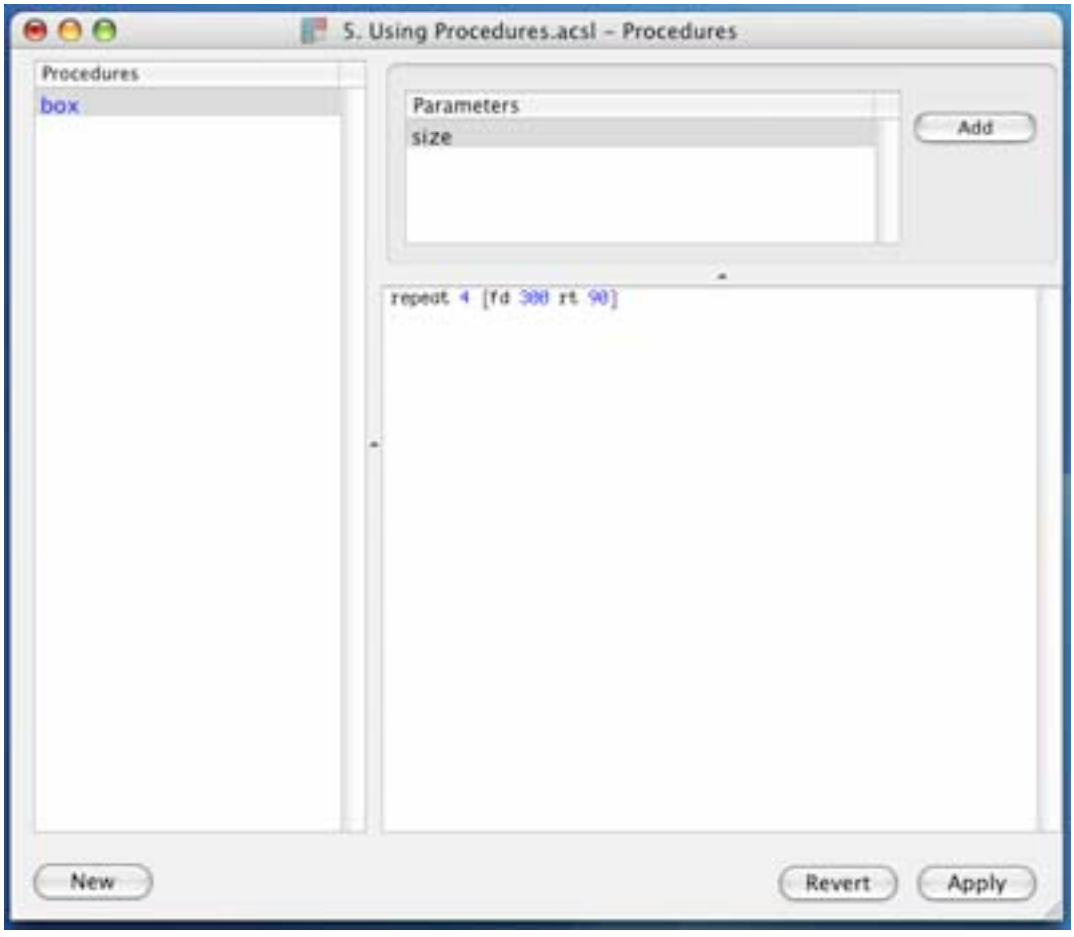
The command to draw all the squares in a circle now becomes:

cs repeat 36 [box rt 10]

which is a bit easier to understand.

Parameters

The box procedure isn't very good if I need boxes of different sizes. Go back to the procedures window, click on the **Add** button next to the Parameters list box, and overtype *param* in the list box with the word size. You may need to double-click on it.



This will be our parameter, and is like a variable (I could have called it anything, but this is meaningful). Change the body of the procedure to refer to the parameter:

repeat 4 [fd :size rt 90]

Note that the name of the procedure has become blue because we've made some changes. Click on the **Apply** button to activate them.

Now to create a square of say, 150 pixels:

cs box 150

and our 'rotating square' command becomes:

cs repeat 36 [box 150 rt 10]

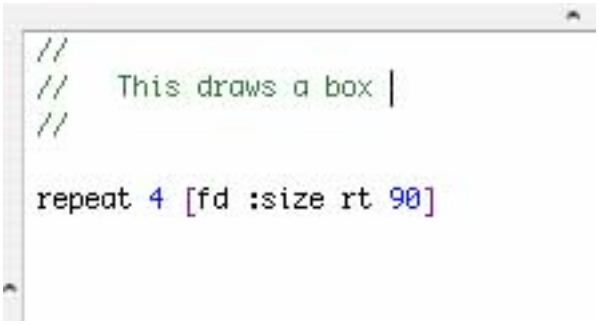
We can now use a variable to change the size of the box as we go:

cs make "amount 30

repeat 36 [box :amount rt 10 make "amount :amount + 10]

Comments

You can add comments to your procedure to remind yourself of what it does. Precede your comments with a double-slash like so:



Note that comments appear in green, numbers are blue, and other text elements are coloured to make it easier to read what's going on. If you don't like it, you can turn off this syntax coloration in preferences.

Local Variables

You can declare a local variable in a procedure using the **local** keyword. This variable is known only inside the procedure, but otherwise can be treated like other (global) variables:

local "lvar

make "lvar 10

make "lvar :lvar + 1

If there is a global variable of the same name, this is masked by the local variable whilst inside the procedure. This stops the procedure from changing variables external to itself which just happen to have the same name as its local ones. Using local variables is good practice as it saves a procedure having unwanted side-effects.

Outputting Results

You can return a value from a procedure. Create a new procedure called **TimesTen**. Give it a parameter called **num**. Make the body of the procedure:

output :num * 10

The procedure should output the value of the input number times ten. Try it out:

TimesTen 50

You should get the value **500**.

Recursion

A procedure can call itself.

Let's consider the mathematical function **factorial**. **factorial(4)** gives the result of **4 x 3 x 2 x 1**. **factorial (5)** gives **5 x 4 x 3 x 2 x 1**, etc. **factorial (0)** returns **1**.

Stating this recursively, factorial(0) = 1, factorial(n) = n * factorial(n-1)

To implement this, create a new procedure called **factorial**. Give the procedure a parameter called **num**, and make the body of the procedure:

```
if :num = 0 [output 1] [ output :num * factorial :num - 1]
```

If the input parameter is zero, **factorial** returns 1, otherwise it returns the input number times the factorial of the next number down.

Try these:

```
factorial 3
factorial 0
factorial 10
factorial 5
```

While and For

By using recursive procedures as in the previous section, you can control how many times you execute the procedure — you don't need to decide how many iterations at the start, which you do with the **Repeat** statement. This means you don't need a **while** or **for** statement.

However, they can be useful, and if you want to, you can roll your own.

Let's create a **while** procedure. The call for the procedure will look like this:

while [*condition*] [*actions*]

so the procedure takes two lists. Create a procedure called **while**, and give it two parameters called **condition** and **actions**.

The start of the body of the procedure should look like this:

```
local "result
make "result run :condition
```

The first line is declaring a local variable called result. This is set on the next line to the result of running the contents of the condition parameter. So, if condition was:

```
[5 > 4]
```

result would be set to true.

The rest of the procedure tests the value of result in an **if** statement:

```
if :result
  [
  run :actions
  while :condition :actions
  ]
  [
  ]
```

If **result** is **true**, the **if** statement runs the statements in the body parameter, and then calls itself recursively with the same parameters. If **result** is **false**, it does nothing so the procedure terminates.

Try out the **while** procedure by running the following two statements:

make "a 55

while [:a > 50] [print :a make "a :a - 1]

So let's have a go at creating a **for** procedure. We want it to look like this:

for *variable-name start-value end-value* [*actions*]

so a call might look like this:

for "i 1 10 [print :i]

so the procedure would set the variable i to 1, 2, 3, etc., up to 10 and for each of those values would print the variable.

So create a procedure called for with parameters **var**, **start**, **finish**, and **actions**.

The first thing the procedure must do is set the variable **i** to the value of the **start** parameter. Since the name of the variable is held in the **var** parameter, this is easy:

make :var :start

Then we need to test if we've finished in an **if** statement:

```
if NOT :start > :finish
  [
  Run :actions
  for :var :start + 1 :finish :actions
  ]
  [
  ]
```

If we haven't finished, we run the actions, then call the **for** procedure recursively with **start** incremented by 1.

Apply the procedure then try this:

for "i 1 10 [print :i]

Thing

Consider a procedure which will make some change to a variable — let's call it **increment**. When we pass the name of a variable to it, it increases the value of the variable by 1. So:

make "i 20

increment "i

would set variable **i** to 21.

Set up a new procedure **increment** with an input parameter called **var**.

For the body of the procedure, we can't say:

make :var :var + 1

because that would evaluate to:

make "i "i + 1

What we need is some way to get the value held by the variable whose name is held by **var**. Logo provides a function, **Thing**, to do just that.

Make :var (Thing :var) + 1

Thing and its argument need to go in parentheses otherwise Logo tries to add 1 to **:var** first.

Importing Procedures

If you've procedures in one Logo document that you want to copy into another, you can simply drag and drop from one document into another.

Select the procedures you want to copy, then drag them into the procedures window of the other document.

Graphics

We've already covered the basic drawing commands in earlier chapters — **Forward**, **Back**, **Right** and **Left**. The chapter on the Turtle covered the Turtle, the Canvas and x- and y-co-ordinates, and the Pen. We'll cover some further graphics topics in this chapter.

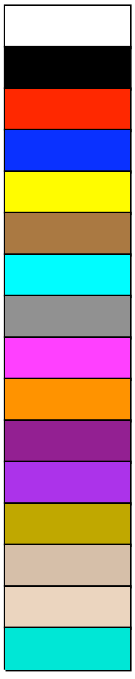
Colours

There are two 'current' colours that drawing can take place with — the Pen Colour, or Foreground Colour, which is used for almost all drawing commands, and the Background Colour, which is used to fill the Canvas when you issue **Clean** or **Clearscreen**.

You can see what these are set to by issuing the **PenColour** and **Background** commands:

```
PenColour
1
Background
0
```

The numbers you get back don't tell you much about what the colours actually look like. They are not colours themselves, but indexes into a colour list:



This is the colour list when ACSLogo starts up. The first colour is Colour 0 (zero) at the top. The last in this list is Colour 15. Beyond that, the colour numbers just repeat the first 16, so colours 16 – 31 are the same as colours 0 – 15, etc. However, you can set any colour number to any colour you like, so this should not be restrictive.

The colour list is held separately for each document.

When you open a document or create a new one, the colour list is set as shown, and the background colour is set to entry zero (white) and the pen colour to one (black).

The colour of each entry in the colour list is specified by how much red, green and blue it contains. So a completely red colour contains 100% red, no green, and no blue. A turquoise-ish colour might contain no red, 60% green, and 60% blue, while black contains no red, green, or blue.

Because all colours are specified in terms of red, green, and blue, this system is known as the **RGB** system. In ACSLogo, rather than specify amounts as percentages, you specify them as a number between zero and one. So the RGB values for the red colour would be (1.0, 0.0, 0.0), and

for the turquoise-ish colour (0.0, 0.6, 0.6). Black is (0.0, 0.0, 0.0).

Here are some examples of colours and their RGB values.

Colour		R	G	B
Red		1.0	0.0	0.0
Green		0.0	1.0	0.0
Blue		0.0	0.0	1.0
Black		0.0	0.0	0.0
White		1.0	1.0	1.0
Pink		1.0	0.7	0.7
Grey		0.5	0.5	0.5
Yellow		1.0	1.0	0.0

So, if I want to draw a line in red, I set the Pen colour to 2 by issuing **SetPenColour 2**, and if I want to set the canvas to black, I issue **SetBackground 1** and **ClearScreen**.

What if a colour I want is not in the list? That's easy enough — just set one of the list entries to the RGB values you require using **SetRGB**, which takes two parameters, a colour number and a list of RGB values:

SetRGB 3 [1.0 0.7 0.7]

This particular example sets pen colour 3 to pink. You then need to issue **SetPenColour 3** to use it for drawing, or **SetBackground 3** and **ClearScreen** to fill the canvas with it.

You can query the RGB values for a colour number using command **RGB**:

```
RGB 3
[1  0.7 0.7]
```

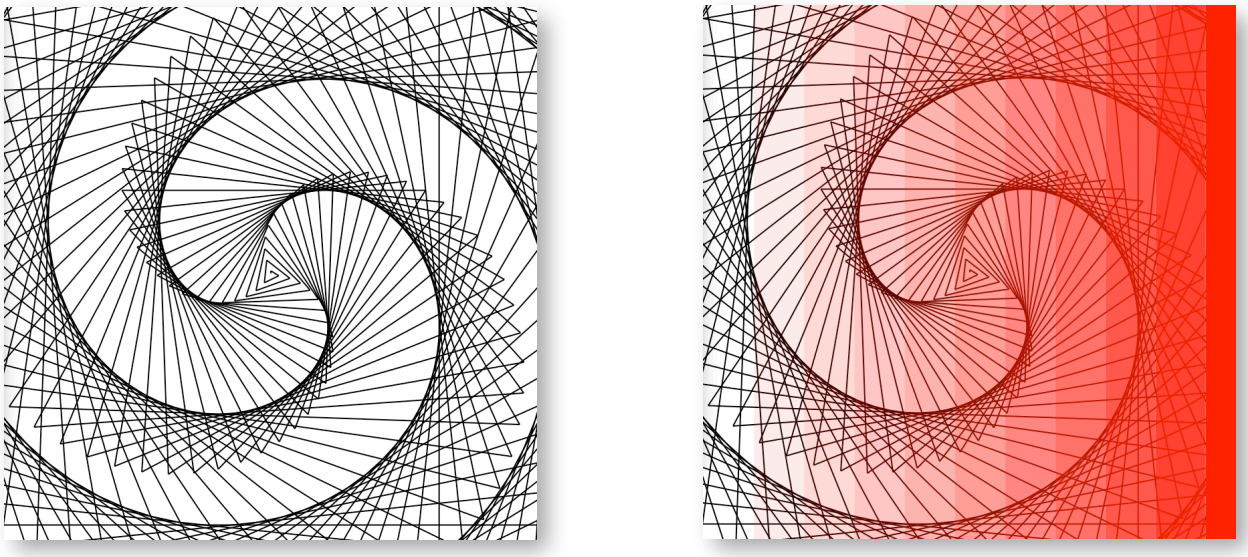
The parameter is the colour number. It returns the red, green, and blue values.

Transparency and Opacity

So far, the lines we've drawn have completely obliterated everything underneath them — the lines are completely opaque. We can make them partly transparent so that what was underneath shows through.

The first diagram here is the spiro example from the Examples file in the install folder. In the

second diagram, I've covered it in strips of an increasingly opaque red colour.

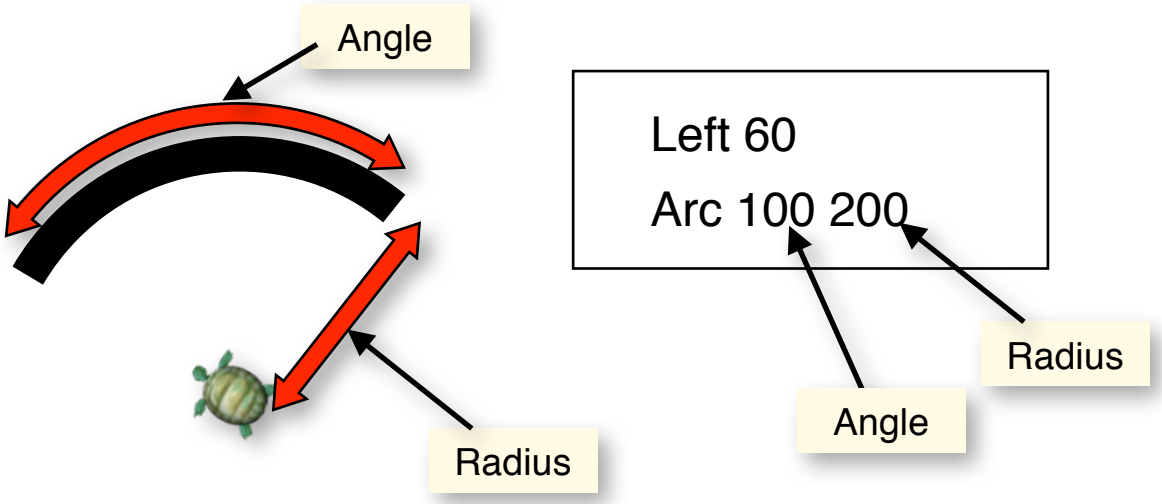


On the left-hand side of the second diagram, the red colour has an opacity of 0.0 – none of the red colour can be seen. It has been set with the command **SetRGB 2 [1.0 0.0 0.0 0.0]** – the fourth entry in the list is the opacity. For the next strip, list values are **[1.0 0.0 0.0 0.1]**, then **[1.0 0.0 0.0 0.2]**, until at the right-hand side, the red colour is completely opaque (**[1.0 0.0 0.0 1.0]**) and none of the picture below it shows through. If you give **SetRGB** three values, Logo assumes that you want a colour that is completely opaque.

Drawing Arcs

The **Arc** command is used to draw circles or parts of circles.

The position and heading of the turtle affect what is drawn. The arc is drawn centred on the turtle's position, starting from a point directly ahead of the turtle, sweeping clockwise. The command takes two parameters, an angle and a radius. Angle is the angle through which the arc is drawn, 360 being a full circle.



Note that Arc does not affect the heading or position of the Turtle.

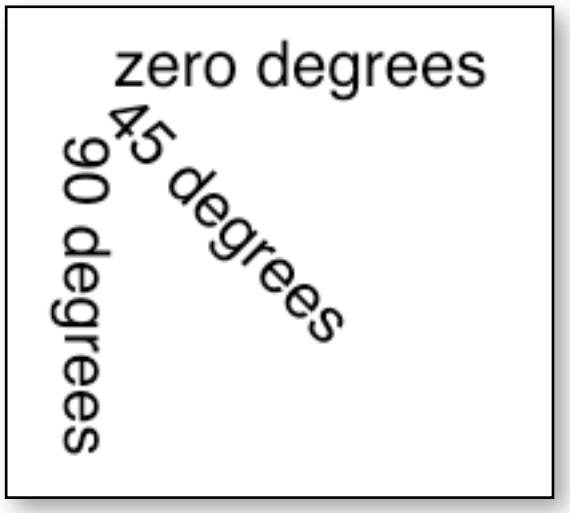
Text

Text is drawn by using the **GraphicsType** command. The command takes a single parameter which can be a word or a list.

The text is drawn at right-angles to the turtle heading.

You set the size of the characters drawn using **SetTypeSize**. These commands demonstrate the sort of thing you can do:

```
SetTypeSize 24
GraphicsType [  zero degrees]
Right 45
GraphicsType [  45 degrees]
Right 45
GraphicsType [  90 degrees]
```



You can also change the font used by **GraphicType**. The command **Fonts** gives a list of fonts available. You can then use any of these entries with **SetFont** to change the current font:

SetFont [AntiqueOlive-Compact]

The **TextBox** command outputs a list describing the size of its parameter if printed by **GraphicsType**. The list is of the form [x y w h], where x,y is the co-ordinate of the bottom-left corner, w is the box width, and h is the box height.

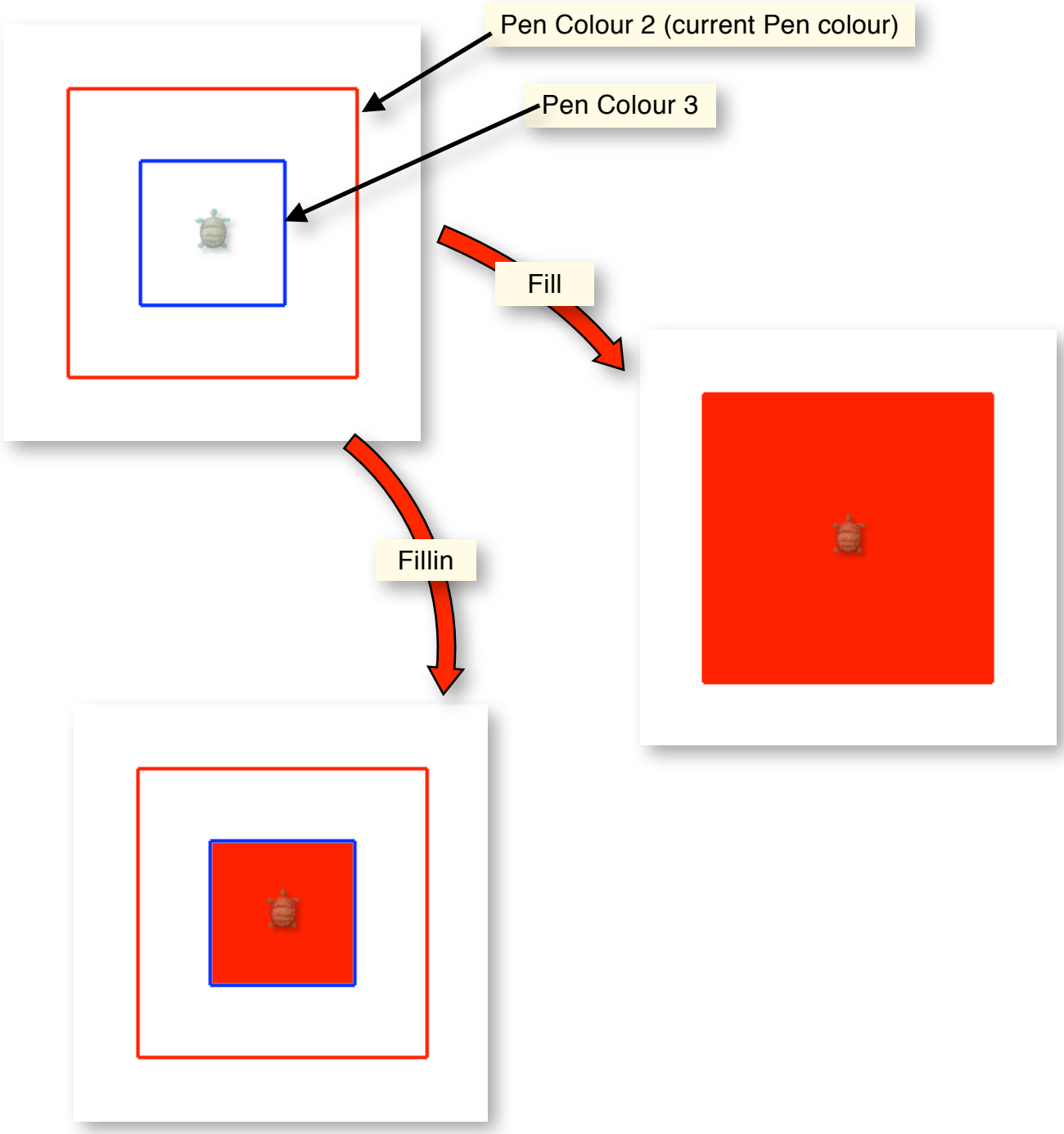
The textbox is not a bounding box - the height of the box is the line-height of the text, and the width includes letter spacing on either side.

Filling Shapes

The squares and other shapes we drew in earlier chapters have been empty – we've just drawn the outline. So how do we fill a shape with colour?

There are two standard Logo commands, **Fill** and **Fillin**.

In the following diagram, I've drawn a square in blue (Pen colour 3) inside a red square (Pen Colour 2). the Turtle is at the centre of both squares.



The current Pen colour is set to 2. If the fill command is issued, it keeps filling (from the current position under the turtle) until it hits a boundary of the current Pen colour — it ignores the inner square which is colour number 3.

Fillin keeps filling until it hits a border which is a different colour from the starting pixel (the one at the turtle position).

There are a couple of problems with these two methods of filling. First, what is filled does not depend just on the shape you've just drawn, but depends on what is on the canvas already, which may be all sorts of stuff depending on what you've been drawing.

The second problem is to do with the way Mac OSX draws lines. To get the beautifully smooth lines and text that you see in OS X, edges of lines can be drawn in a slightly different colour from the colour that you asked for. This makes the lines appear smoother to the eye (and brain). This technique is known as anti-aliasing. It means that the Fill command especially may not recognise a thin border when it hits it.

We'll look at alternative ways to do fills in the Paths chapter.

Shadows

The **SetShadow** command sets a dropshadow for all subsequent drawing.

```
ClearScreen
SetShadow [10 -15 5]
SetTypeSize 324
SetPenColour 2
GraphicsType "S
```

Here the **SetShadow** command makes the shadow drawn be offset 10 pixels to the right, 15 pixels down, and with a blur radius of 5 (the blur radius is a measure of how much the dropshadow is spread). You can add a fourth parameter - pen colour, which specifies which colour to use as the shadow colour.

```
ClearScreen
SetShadow [10 -15 5 3]
SetTypeSize 324
SetPenColour 2
GraphicsType "S
```

To stop any more shadows being drawn on subsequent drawing, call **SetShadow** with the empty list:

```
SetShadow [ ]
```

Paths

We saw in the Graphics Chapter that the **Fill** and **Fillin** commands often don't give you what you want — they fill pixels based on what is already on the canvas.

I've added to ACSLogo a non-standard command called FillCurrentPath. This command doesn't care what is on the canvas already. It just tries to fill the last shape drawn. To understand exactly what it does, we need to understand the concept of the current path.

When you draw a line (using **Forward**, **Arc**, etc.), ACSLogo adds the line to the current path, which is just a list of the lines draw. The command **CurrentPath** returns the contents of the current path as a list of lists:

```
ClearScreen
Forward 200
CurrentPath
[[moveto 0 0][lineto 0 200]]
```

The **ClearScreen** command caused whatever was in the current path to be cleared, and made the Turtle move to x co-ordinate zero, y co-ordinate zero. At this point, nothing has been draw, so the current path is still empty. The **Forward** command made the Turtle move from co-ordinates (0,0) to (0,200). This generated the *moveto* and *lineto* entries in the current path.

As long as the pen stays down (so drawing is done), entries are added to the current path:

```
Right 90
Forward 100
CurrentPath
[[moveto 0 0][lineto 0 200][lineto 100 200]]
```

Issuing **PenUp** clears the current path:

```
PenUp
CurrentPath
[]
```

FillCurrentPath

The main point of keeping a current path is to be able to fill the shape you've just drawn easily. After drawing, say, a rectangle, you can just issue **FillCurrentPath** and it will fill the shape in the current pen colour. With **Fill** and **Fillin**, you have to position the Turtle somewhere in the middle of the shape after drawing it before issuing the command.

StrokeCurrentPath

You can also issue StrokeCurrentPath to draw a line around the path with the current pen width and pen colour.

Saving Paths

Because the current path is just a list, you can save it in a variable:

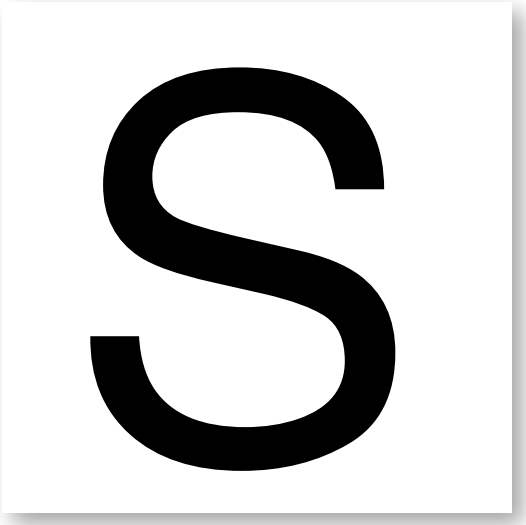
```
Make "p CurrentPath
:p
[[moveto 0 0][lineto 0 200][lineto 100 200]]
```

This can then be used in StrokePath and FillPath which are similar to their current path equivalents, but each takes a path as parameter. The value of this can be seen in the next section.

Text

When you draw text using GraphicsType, the text is added to the current path. By saving it in a variable, we can get some interesting effects. First let's draw a big letter and save it in the current path:

```
SetPenColour 1
ClearScreen PenUp
SetPosition [-150 -150] PenDown
SetTypeSize 400
GraphicsType "S
make "p CurrentPath
```

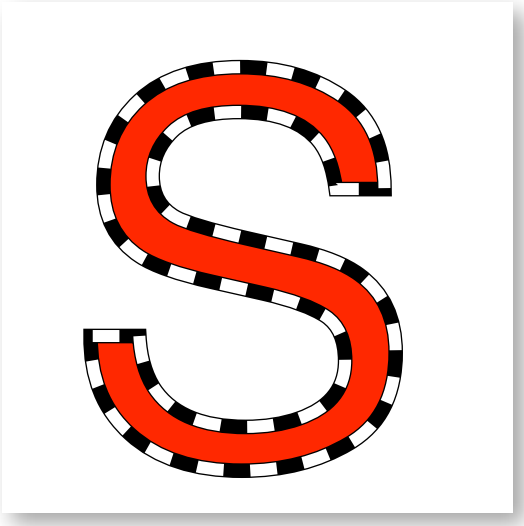


If we issue ClearScreen, it clears the current path, but we have the path saved in a variable which we can fill and then stroke to get an outlined letter:

```
ClearScreen
SetPenColour 2
FillPath :p
SetPenWidth 11
SetPenColour 1
StrokePath :p
```

I can then add a further dash of interest by stroking with a dashed white line:

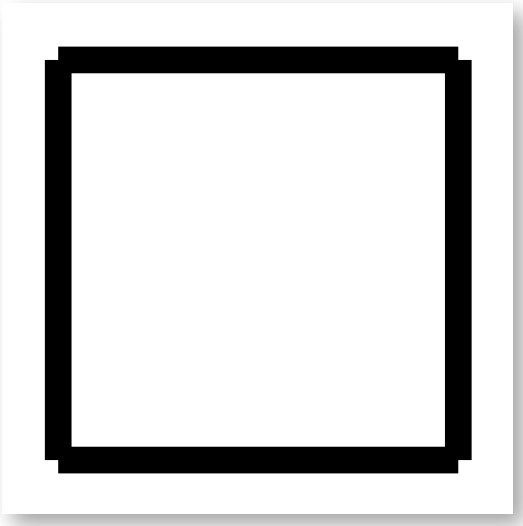
```
SetLineDash [0 20 20]
SetPenColour 0
SetPenWidth 9
StrokePath :p
```



Corners

If you use a very thick pen width when drawing shapes, you may have noticed that the corners are not very neat:

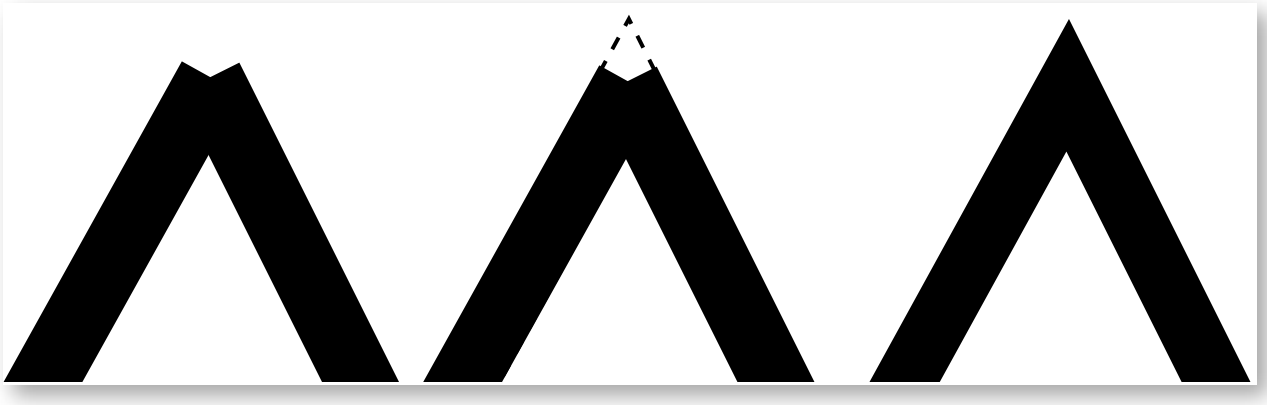
```
SetPenWidth 20
Repeat 4 [ Forward 300 Right 90 ]
```



You don’t notice this with a small pen width. To see what’s happening, I’ll blow things up and make the lines semi-transparent. Here's the top-left corner of the rectangle:

You can see that at the corners, the lines overlap on the inside of the angle, and on the outside of the angle, it looks like there's a bit missing. This is because each line is drawn separately.

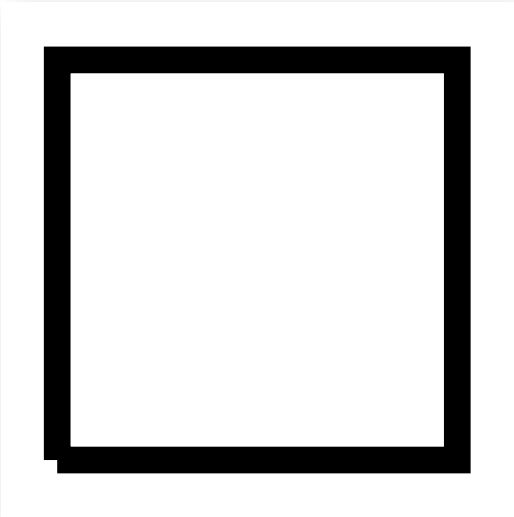
With an acute angle, the problem is even more obvious:



The gap is obvious in the left-hand diagram. The middle one shows that the outside edges can be extended to meet at a point. The right-hand shows what happens when that's done – a much neater join.

Using paths solves the problem – almost.

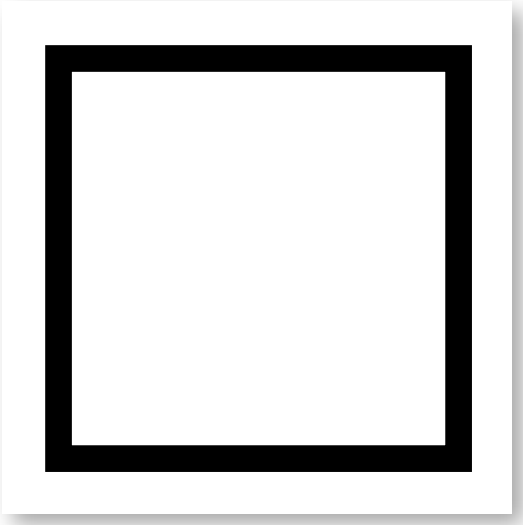
```
Repeat 4 [ Forward 300 Right 90 ]
Make "p CurrentPath
ClearScreen
StrokePath :p
```



You can see that three of the four corners are rendered correctly, but the fourth one – which is where the path starts and ends – is not, because as far as ACSLogo knows, the lines are not joined.

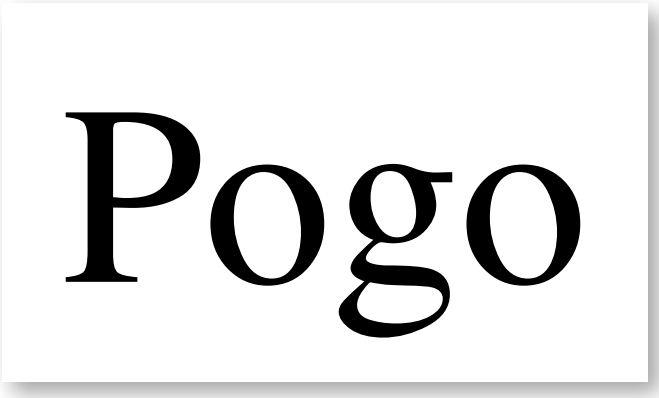
The only way to do that is close the path by adding a close statement to the end of the path:

```
Repeat 4 [ Forward 300 Right 90 ]
Make "p CurrentPath
Make "p LastPut [close] :p
ClearScreen
StrokePath :p
```



Holes

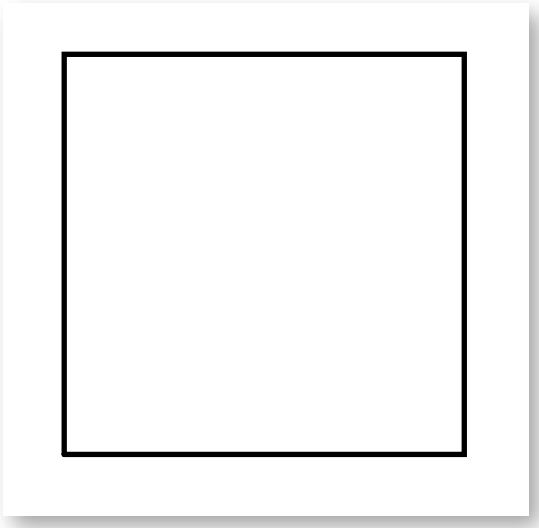
When you look at the letters that make up words, such as the one below:



you can see that some letters contain ‘holes’. Letters are just filled paths. The letter ‘o’, for instance, is just a small ellipse inside a bigger one. When the bigger ellipse is filled, why doesn’t that just fill in the smaller ellipse as well?

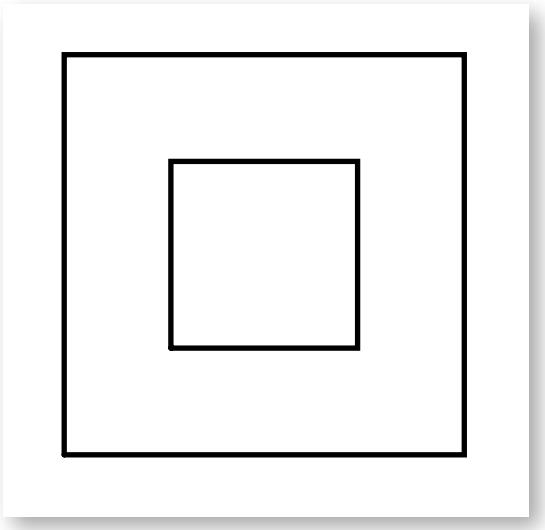
Let’s look at simpler shapes — rectangles — and try a few things out. First I draw a rectangle and save the path in a variable called *bigrect*:

```
SetPenColour 1
Repeat 4 [Forward 150 Right 90]
Make "bigrect CurrentPath
```



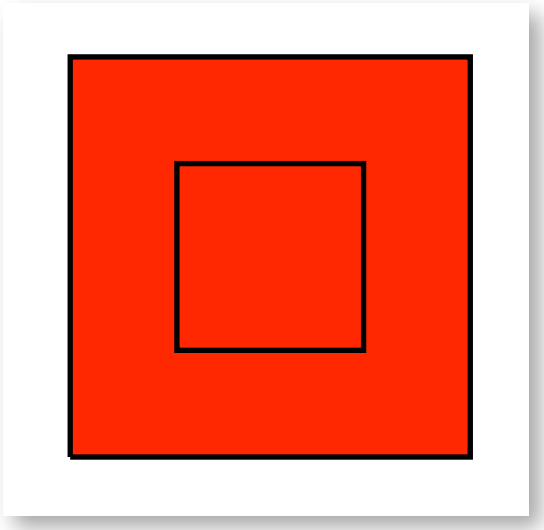
Then I move in a bit and draw a smaller rectangle and save its path in a variable called *smallrect*:

```
PenUp SetPosition [40 40] PenDown
Repeat 4 [Forward 70 Right 90]
Make "smallrect CurrentPath
```



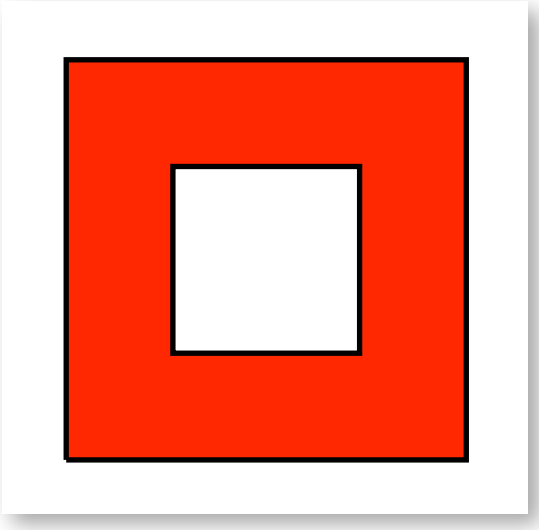
Next I join the two paths together with the **Sentence** command, which joins two lists – this is to make the two paths into one object – then fill the path and stroke it:

```
Make "p1 Sentence :bigrect :smallrect
SetPenColour 2 FillPath :p1
SetPenColour 1 StrokePath :p1
```



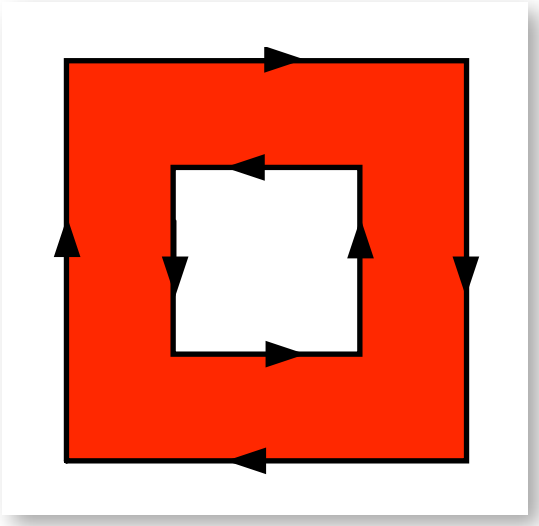
You can see that the fill has just filled in both rectangles - there's no ‘hole’. This time I’ll draw the small rectangle in a different way — first I’ll go right, then draw the rectangle going anti-clockwise:

```
PenUp SetPosition [40 40] PenDown
Right 90 repeat 4 [fd 70 Left 90]
Make "smallrect CurrentPath
Make "p1 Sentence :bigrect :smallrect
SetPenColour 2 FillPath :p1
SetPenColour 1 StrokePath :p1
```



This time we’ve got the hole — so holes in overlapping shapes are dependent on which directions their paths ‘wind’.

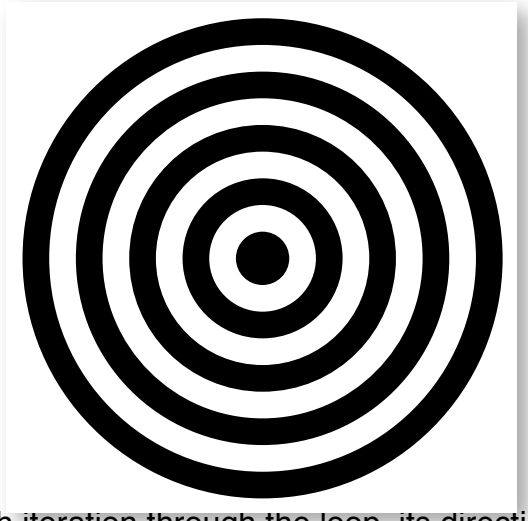
You can see the directions of the paths in this diagram:



The outer rectangle's path goes clockwise, the inner one anticlockwise.

This nesting can go on indefinitely, with each path going in the opposite direction to the one outside it:

```
Make "radius 180
Make "a1 [ ]
Repeat 9
[
      ClearScreen
      Arc 360 :radius
      Make "a1 Se ReversePath :a1 Se
           CurrentPath [[close]]
      Make "radius :radius - 20
]
ClearScreen FillPath :a1
```

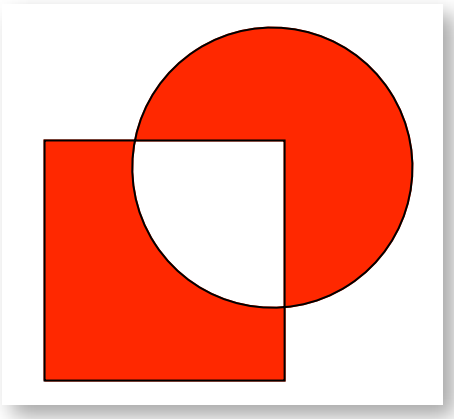


The path is accumulated in a variable called *a1*. Each iteration through the loop, its direction is reversed using command **ReversePath**, and it's concatenated with the current path (which is the arc just drawn). This means that we end up with a path of circles, each going in the opposite direction to the previous one.

Overlapping objects so that one does not lie entirely within the other gives us more scope for creativity.

The complications in the code are mostly due to positioning prior to drawing the square and

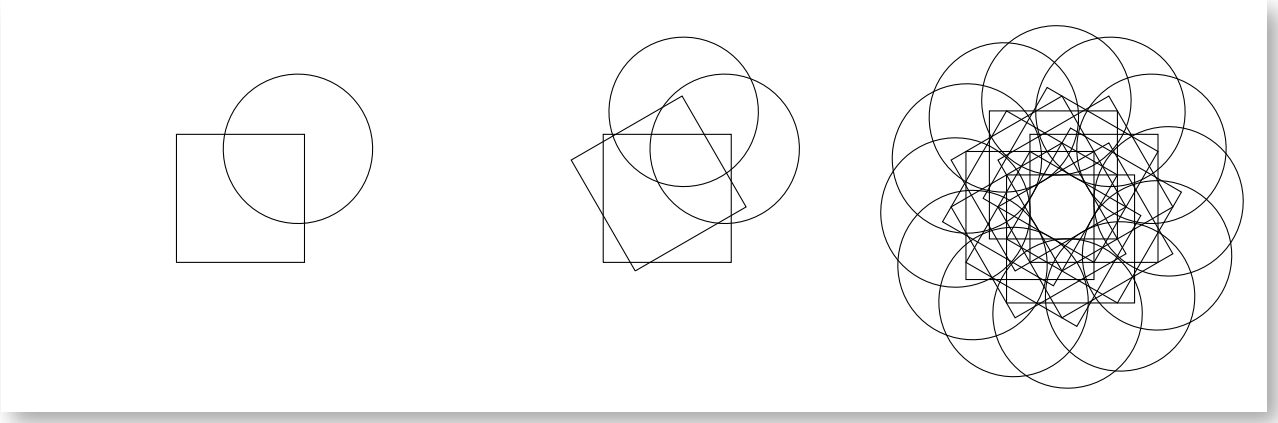
```
PenUp right 30 Back 60 Left 30 PenDown
Repeat 4 [Forward 120 Right 90]
Make "a1 LastPut [close] CurrentPath
PenUp Right 30 Forward 60 Left 30
Right 57 Forward 100 PenDown Arc 360 70
Make "a2 reversepath LastPut [close] CurrentPath
PenUp Back 100 Left 57
Make "a2 Se :a1 :a2
SetPenColour 2 Fillpath :a2
SetPenColour 1 StrokePath :a2
```



the circle, as I want them offset from the canvas origin and from each other. Also, I've deliberately used relative commands to do the positioning (Left, Right, Forward, Back) rather than absolute commands (SetPosition, SetHeading).

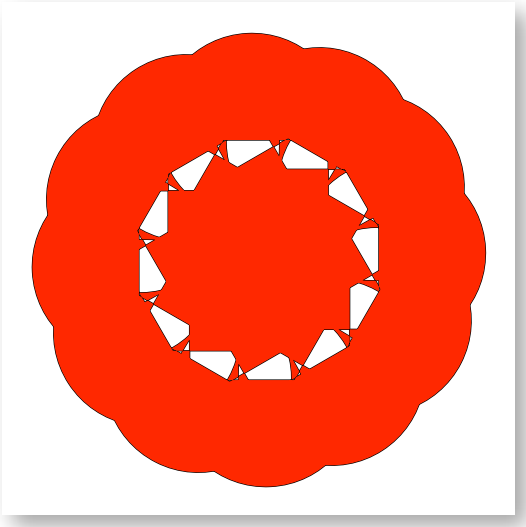
To simplify following code, I've put all that code in a procedure called shape and replaced the last three lines with **Output Se :a1 :a2**. So the procedure outputs the path and doesn't draw it.

In the following diagram, the drawing on the left shows the result of stroking the output from the *shape* procedure. For the middle drawing, shape is called again after the Turtle has turned left 30 degrees. For the last drawing, that has been repeated another ten times.



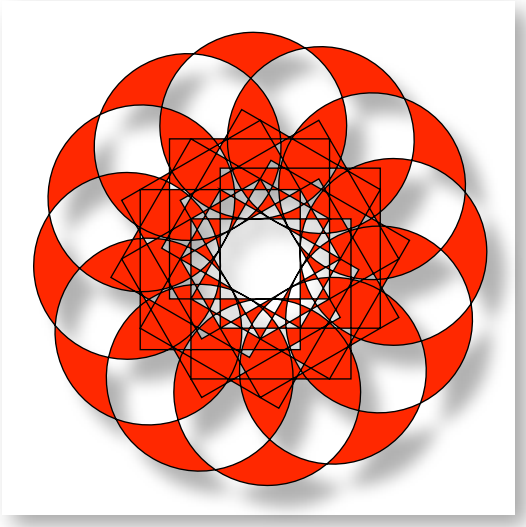
If I fill the shape, I just get a big blob:

```
Make "p [ ]
Repeat 12
[
    Make "p Se :p shape
    Left 30
]
ClearScreen
SetPenColour 2 FillPath :p
```



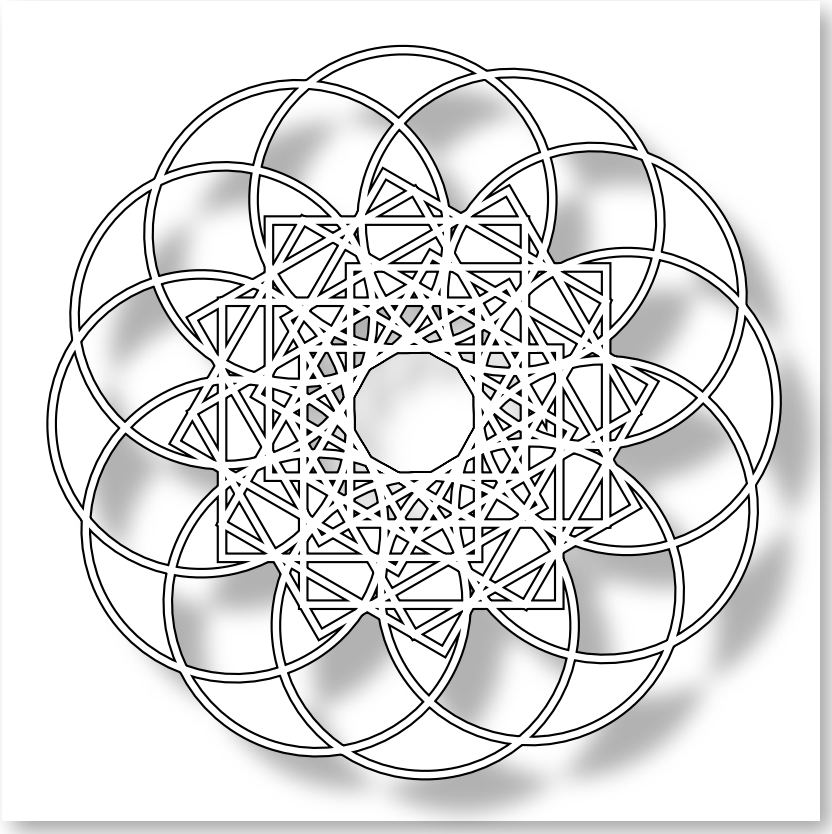
Reversing the path each time through the loop gives us some more interesting holes:

```
Make "p [ ]
Repeat 12
[
    Make "p Se ReversePath :p shape
    Left 30
]
ClearScreen
SetShadow [ 15 -15 11 ]
SetPenColour 2 FillPath :p
SetShadow []
SetPenColour 1 StrokePath :p
```



Note that I've added a shadow before doing the fill — this helps show that the holes are really holes and not just white-coloured bits of the pattern.

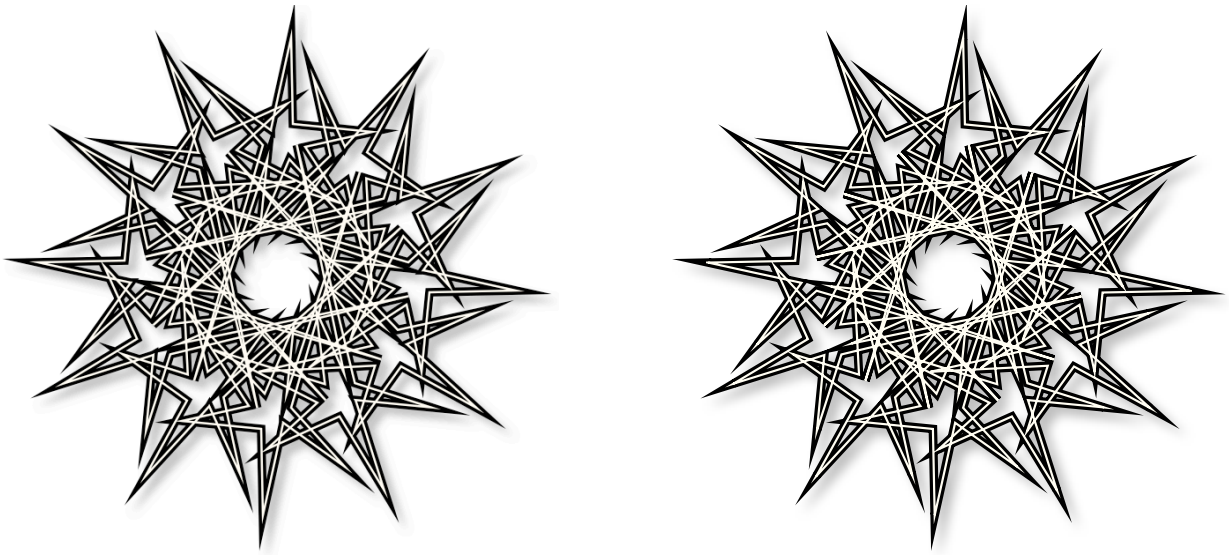
Finally, this is the same pattern, but the fill has been done in white, then a stroke has been done with a black line, then another stroke with a thinner white line over it.



Vector Graphics

The paths we've been talking about are examples of *vector graphics* — they store the image as a collection of lines (straight and curved) rather than as tiny rectangles of colour (pixels) which bitmap images are stored as.

The advantage of a vector-based image is that it can be magnified indefinitely without loss of quality – with a bitmap-based image, the more you magnify it, the more you will see the pixels making it up – like sticking your face against a TV screen. If you're viewing this from a PDF file on a computer, you can see this for yourself by looking at these seemingly identical images:



Zoom in on the page using the zoom button in the program you're viewing this with. Keep zooming in , and after two or three zooms, you should see that the image on the left starts to look pixellated, while the one on the right remains sharp. The left-hand image is a bitmap, while the one on the right is a vector graphic.

If you're reading this on a printed page, I'll have to do the zooming for you – I'll choose the last letter in this paragraph.

See how much better quality the zoomed vector graphic image is.

Exporting Vector Graphics

If you want to save an image you've slaved over in ACSLogo, you need to export it. You can export it as a bitmap (**Export/Graphics**) or as vector graphics (**PDF** or **SVG**). You need to decide whether you want the ease of use of a bitmap to say, embed in a web page, or a high-quality stand-alone vector graphic – a PDF; or a high quality vector graphic which can be embedded in a web page if you know what you're doing, but isn't viewable by all browsers – SVG.

When you export a bitmap, you get exactly what you see on the canvas (apart from the Turtle). When you export vector graphics, you get the path version of the graphics, like you get with the **CurrentPath** command. ACSLogo keeps all the paths that have been drawn since the last

ClearScreen command. When these are exported, the image is subtly different from the image on the canvas. Just like when using **StrokePath** and **StrokeCurrentPath**, line-joins will be neater, and the results of **Fill** and **FillIn** commands will not be exported.

Files

In this tutorial, I'll be looking at how you can read from and write to files within ACSLogo. You need a fairly basic understanding of the OSX file system — how files exist within directories, that sort of thing.

There are two types of commands that deal with files — file manipulation commands, which are used to open, read and write files, and file management commands, which give you information about the file system and let you change your location within it. We'll cover the latter commands first.

File Management Commands

These commands are simple versions of Unix shell commands that you can use in the Terminal application.

First of all, you need to know where you are in the file system — this position is the current working directory, and is the place where any files you write will be written to and any files you read will be read from.

The command for this is **pwd** (this is the same name as the unix command and stands for *print working directory*).

When I issue the command, I get:

```
pwd
/Users/alan/Documents
```

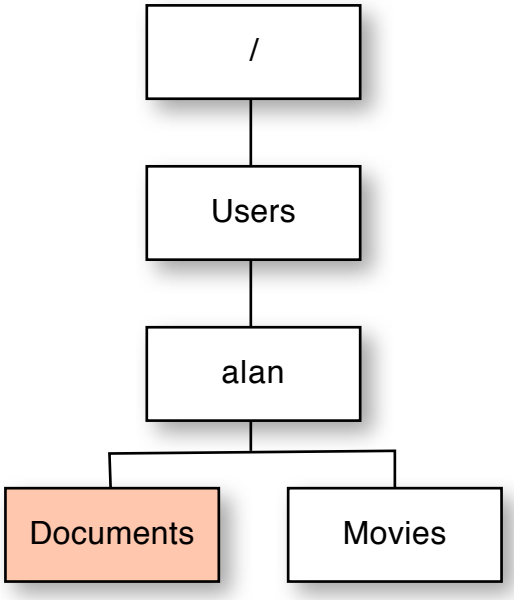
See what happens when you issue the command— you should get something different.

Since **pwd** is just outputting a string, you can store it in a variable:

```
make "temp pwd
:temp
/Users/alan/Documents
```

To change your position within the file system you can use the **CD** (change directory) command. **CD** takes one parameter, which can be a string or a list.

Consider the following file system. At the moment, the current directory is /Users/alan/documents:

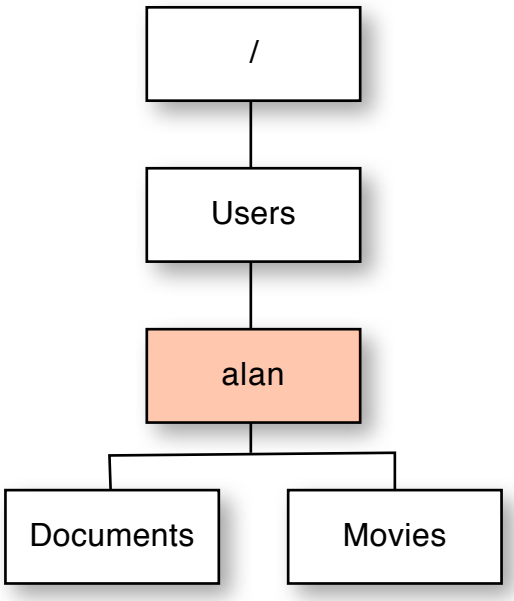


I can use the CD command to change my position. If I issue:

CD ".."

(two full stops or periods) I go up a level. Remember that the parameter to **CD** has to be a string or a list.

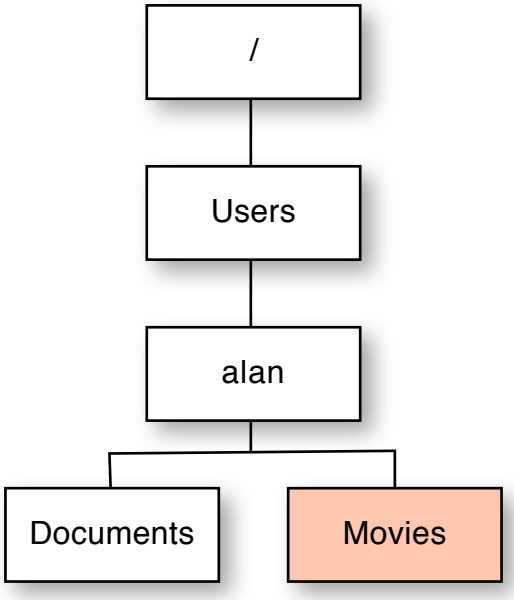
I'm now positioned here:



If I now issue:

CD "Movies

my current directory changes to Movies:



Rather than go up and down the directory hierarchy incrementally, I can specify a complete path — say I want to move to the Documents directory, the full path is /Users/alan/Documents. If I try and give that as a parameter to CD, I get an error:

```
CD "/Users/alan/Documents
unknown function Users
```

This is because the / character, which is used to delimit directory names, is interpreted by the ACSLogo parser as a divide operator. To get the full pathname input to CD without being tampered with, I need to put it in a list:

```
CD [/Users/alan/Documents]
pwd
/Users/alan/Documents
```

Another useful thing to know is that as shorthand for the home directory (in my case /Users/alan), you can use a tilde (~). This can be used on its own or as part of a path:

```
CD [~/Movies]
pwd
/Users/alan/Movies
```

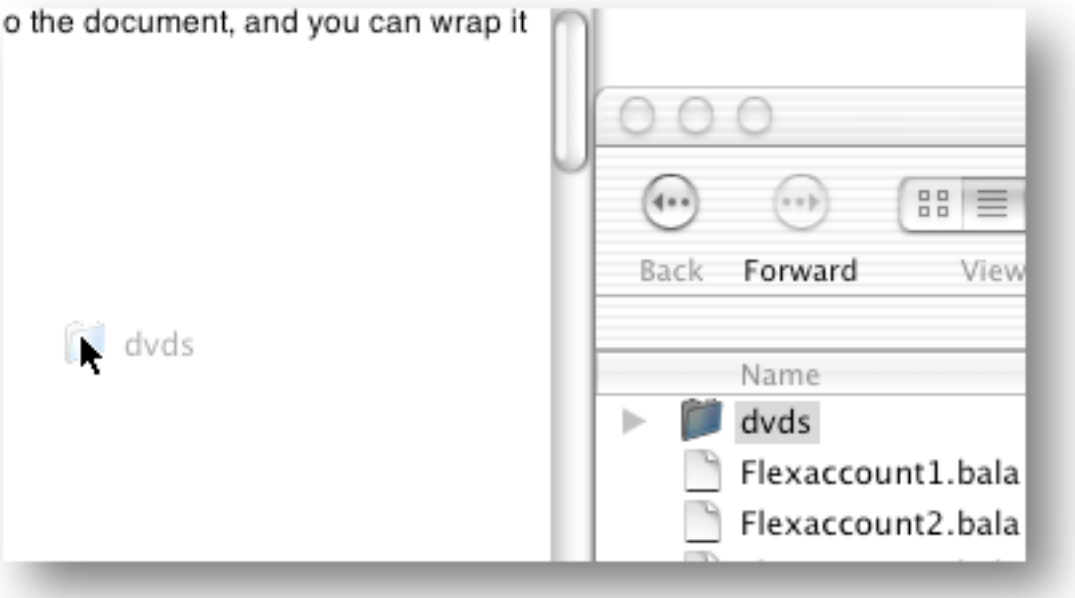
If you give CD an empty string or empty list as input, it will make the home directory the current directory:

```
CD "
pwd
/Users/alan
```

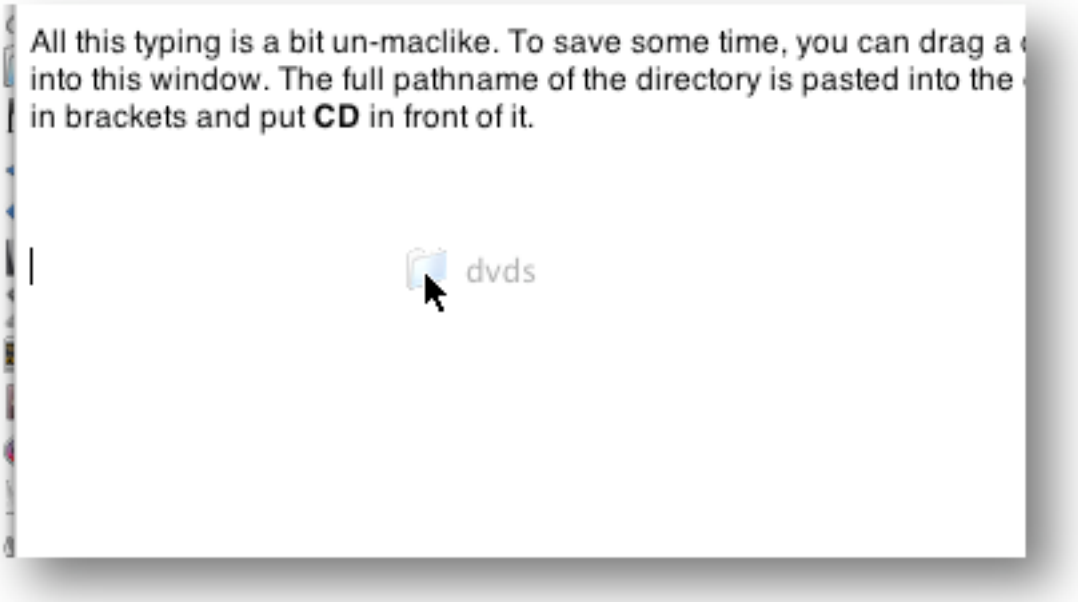
All this typing is a bit un-maclike. To save some time, you can drag a directory from a finder window into the ACSLogo Main window. The full pathname of the directory is pasted into the document, and you can wrap it in brackets and put CD in front of it.

These are the steps:

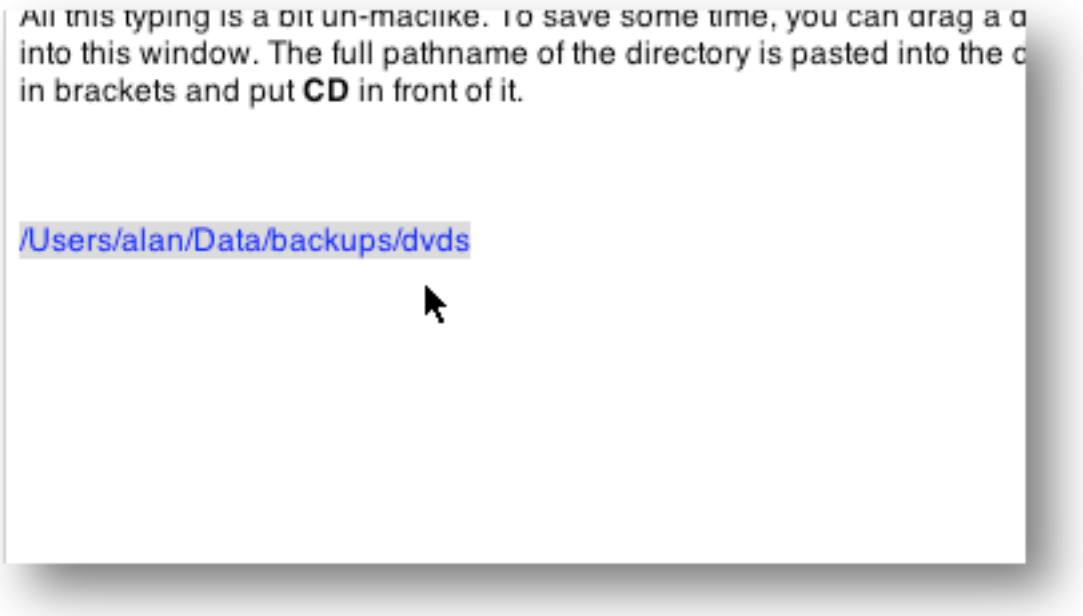
1. **From a finder window, drag a folder over the ACSLogo Main window:**



2. **A cursor will appear. Position it where you want to drop the directory name, and press Control:**



3. **Release the mouse button, and the full path name will appear:**



To see what files are in the current directory, use the **Dir** command. This lists the files one per line. So that you can distinguish directories, each one is followed by a /:

```
pwd
/Users/alan/Documents
Dir
.localized
dabsOrder.pdf
DVLA/
edp24.pdf
Einstein.doc
Einstein.pdf
FAXstf X User Data/
file1.graffle
Installer Logs/
Microsoft User Data/
ms.pdf
YarisRegistration.jpg
```

File Manipulation Commands

In this part, we'll look at writing to files and reading from them. The process for writing to a file is:

- Open a file for writing
- Write data to the file
- Close the file

The process for reading from a file is:

- Open a file for reading
- Read data from the file
- Close the file

Let's look at writing to a file first.

To open a file for writing, issue **OpenWrite**. This takes a parameter which can be a string or a list:

OpenWrite "testlogo.txt

Issue the command above to open the file in the current directory.

There are a number of commands for writing data into the file. These are all variations of commands we've seen already, prefixed by an **F**:

FPrint

FType

FShow

Let's write some data to the file. Issue these commands:

FPrint [the end]

FPrint [of the world]

FPrint [is nigh.]

Then we need to close the file. Issue this command:

CloseWriteFile

Now go into the OSX Finder and find the file **testlogo.txt**. If you double-click on the file, it should open in TextEdit. Check that the contents match the **FPrint** statements above.

So much for writing to a file. Now we'll try reading from one. We'll use the file we've just written. To open it, issue:

OpenRead "testlogo.txt

To read a line from the file into a word, issue:

FReadWord

To read a line from the file into a list, issue:

FReadList

If you keep issuing **FReadList**, you eventually get an empty list back. To close the file, issue:

CloseReadFile

Movies

The algorithmic drawings we can do in logo respond very well to being animated. With animation, we can change the items we've drawn with respect to their location, colour and shape over time.

With examples like the Sierpinski triangle which vary according to an input parameter (level), animation can show the variation in the triangle as the level is changed.

All animations consist of a sequence of images or frames. ACSLogo uses the **Snap** command to capture each frame.

Animation in ACSLogo

The process is:

Create a new movie file

Repeat several times:

Draw an image

Snap the image

Close the movie

You'll probably need to **ClearScreen** before you draw each frame.

An Example

Let's work through an example. What we'll draw in this animation is an outlined letter. The animation will show how the shape of the letter changes as we increase the thickness of the outline.

The size of the each animation frame will be the size of the graphics window. It's probably too big at the moment. Using the **Special/Canvas Size…** menu, set the canvas size to 400 pixels wide and 400 pixels high.

First we need to set the size of the type. Execute this command:

SetTypeSize 250

Then let’s try drawing a capital 'S':

GraphicsType "S

Let's move the turtle a bit to make the letter more central:

SetPosition [-100 -100]

I can do a **Clean** command to clear the screen without moving the turtle, then draw the 'S':

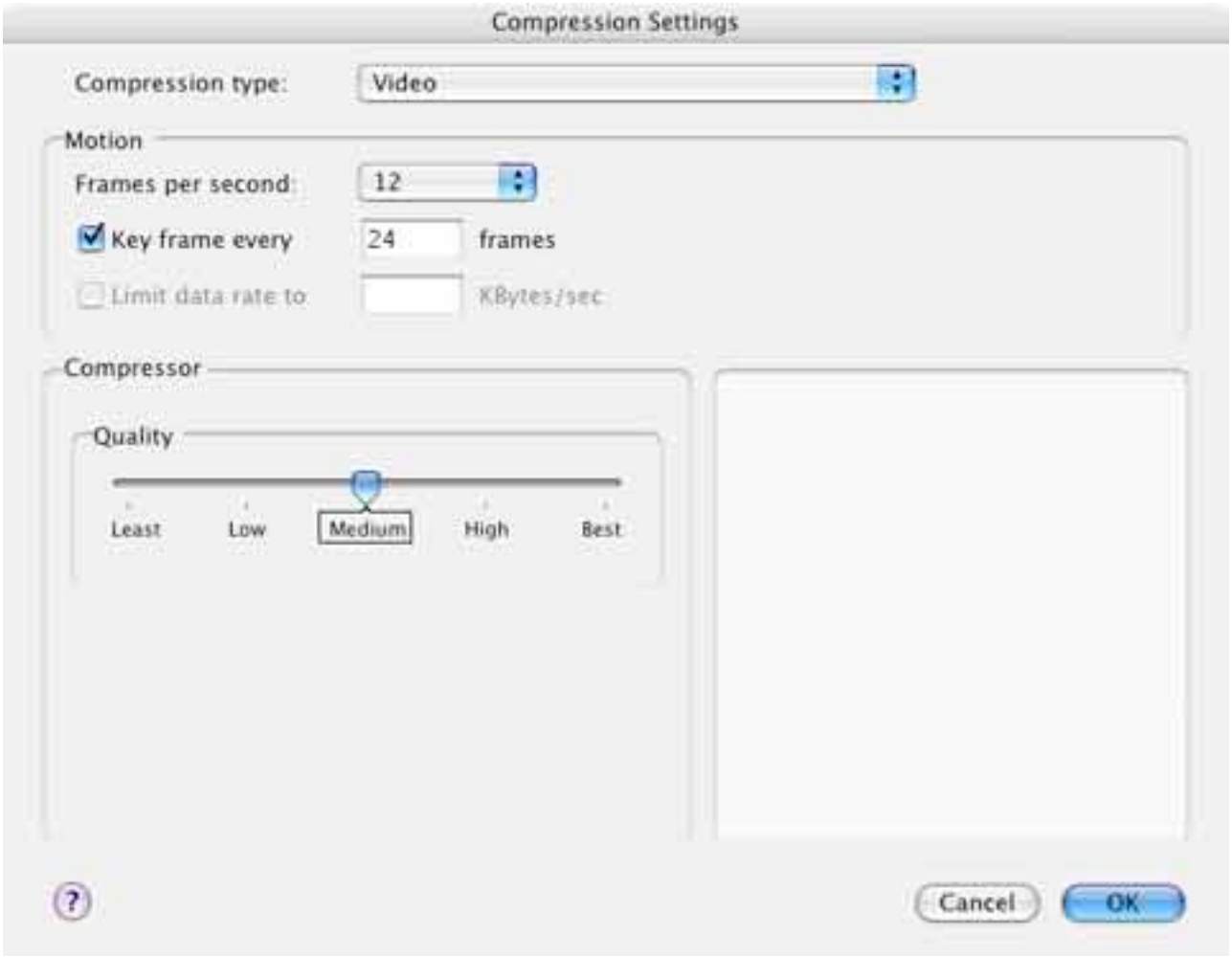
Clean GraphicsType "S

now I'm going to combine these with commands to draw the letter in colour 2, then outline it in colour 1:

Clean SetPenColour 2 GraphicsType "S SetPenColour 1 StrokeCurrentPath

For the animation, I'm going to repeatedly do the strokepath while increasing the pen width each time.

Let's start the movie. Choose **Create Movie…** from the **Special** menu and choose a location and file name to save it to. Then complete the **Compression Settings** window as shown:



We'll first set a variable called pathwidth to 1, and increase it as we go through the movie. Let's do the set-up stuff and **Snap** the first frame of the movie:

```
SetTypeSize 180
SetPosition [-205 -60]
Clean SetPenColour 2 GraphicsType "Logo
Make "pathwidth 1
SetPenWidth :pathwidth
SetPenColour 1
StrokeCurrentPath
Snap
```

Now we'll capture all the frames. Lets capture 60 of them. For each frame, we'll do a **StrokePath**, then increase pathwidth and call **SetPenWidth**:

```
Repeat 60 [Make "pathwidth :pathwidth + 2 SetPenWidth :pathwidth
    StrokeCurrentPath Snap]
```

Select **Finish Movie** from the **Special** menu. If you hold down the Option (alt) key, you can select menu item **Finish Movie and Open** which opens the movie after saving it. Otherwise, find your movie in the Finder and double-click on it to watch it.

Speech & Music

OS X has the ability to synthesize speech, and Quicktime has the ability to play synthesized musical instruments. ACSLogo has commands which can tap into both of these capabilities. In this chapter, it's a good idea to copy and paste the commands into the ACSLogo main window to try them out for yourself.

Speech

Speech is fairly simple. You use the Say command. Give it a word or list as its parameter. in general it's easier to use a list. Issue this command:

Say [Hello there]

You can change the voice of the speech synthesizer using **SetVoice**.

SetVoice [com.apple.speech.synthesis.voice.Albert]

Say [Hello there]

Issue the **Voices** command to get a list of available voices:

Voices

Use **Voice** to show the current voice:

Voice

When you issue the **Say** command, the command finishes straight away, which the speech is still being said. This means that two **Say** commands in quick succession will cause the second command not to be spoken, because the first is still being said. To show this, issue these two together:

Say [The end of the world]

Say [is nigh]

To wait until the previous **Say** command has finished, use **WaitForSpeech**:

Say [The end of the world]

WaitForSpeech

Say [is nigh]

Music

Music is more complicated. the main ACSLogo command is **Play**.

The easiest way to approach this is to look at an example. Issue this command:

Play [1 [60 100 60]]

Play's parameter is a list. The first element of the list is an instrument number — in this case 1, which is a grand piano. To get a list of the instruments available, issue this command:

Instruments

The instrument number input to **Play**, though, can only be a number - it can't be one of the descriptions.

The second element of the list is another list, and this represents a note. The first item in the list, 60, is the duration of the note. The time is measured in ticks, which are sixtieths of a second, so 60 ticks is one second. The second item in the list is the loudness (sometimes called acceleration — how hard the note is hit). In this case it's 100. The maximum is 127. The third item represents the note itself — 60 is middle C. The notes are numbered like keys on a piano keyboard, including the black notes, so E is 64, G is 67, and C of the next octave up is 72.

To play a chord rather than a single note, add other notes to the end of the note-list:

Play [1 [60 100 60 64 67 72]]

To play the notes one after the other rather than simultaneously, each note requires its own note-list:

Play [1 [60 100 60][60 100 64][60 100 67][60 100 72]]

To play more than one instrument at a time, group the list for each instrument into another list. In the following example, the first instrument is a grand piano (1), the second instrument is a woodblock (116) playing twice as fast. Select all the lines and execute:

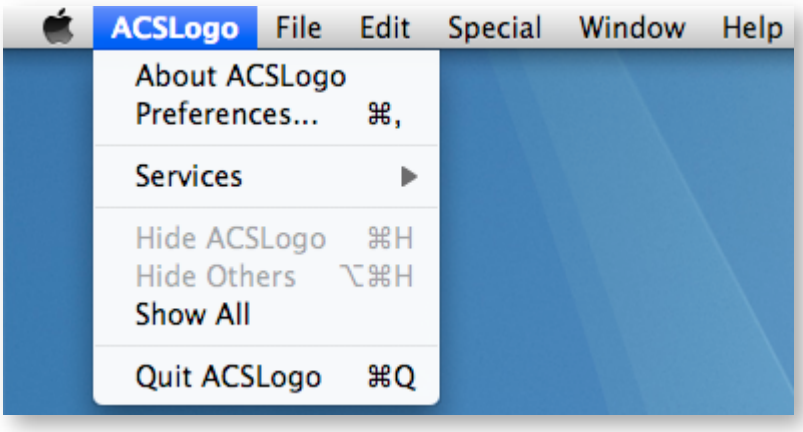
Play [[1 [60 100 60][60 100 64][60 100 67][60 100 72]]

[116 [30 100 60] [30 100 60] [30 100 60] [30 100 60] [30 100 60] [30 100 60] [30 100 60] [30 100 60]]]

Appendix A: Menus

The ACSLogo Menu

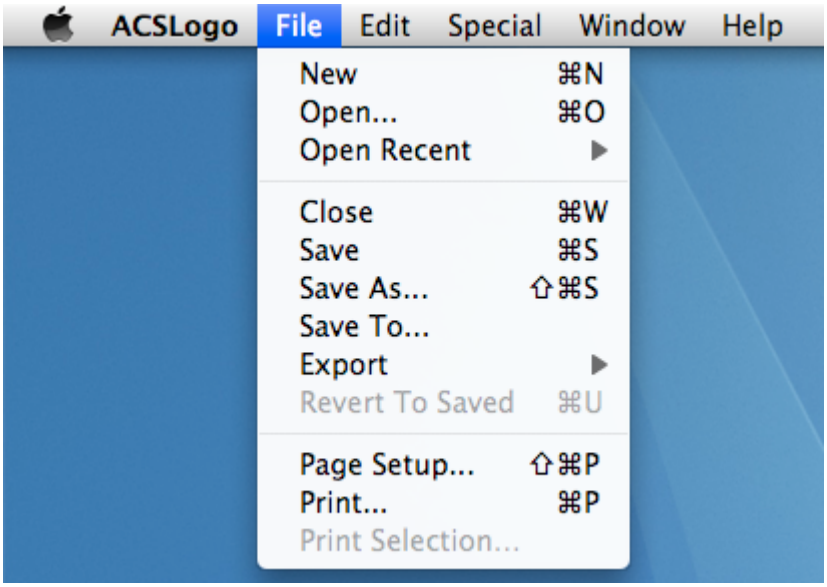
The **ACSLogo** menu contains the standard OSX entries. For more details on **Preferences**, see Appendix B.



The File Menu

The **File** menu contains the standard OSX entries:

New	creates a new untitled document
Open…	shows a dialog to open an existing ACSLogo file
Open Recent	contains a submenu of recently edited documents
Save	saves the current document
Save As…	saves the document under a new name



Save To…	saves a copy to another name
Export	expands to a submenu - explained below.
Revert To Saved	restore the document to the last saved version
Page Setup	choose printing attributes
Print…	print the document
Print Selection…	print selected text

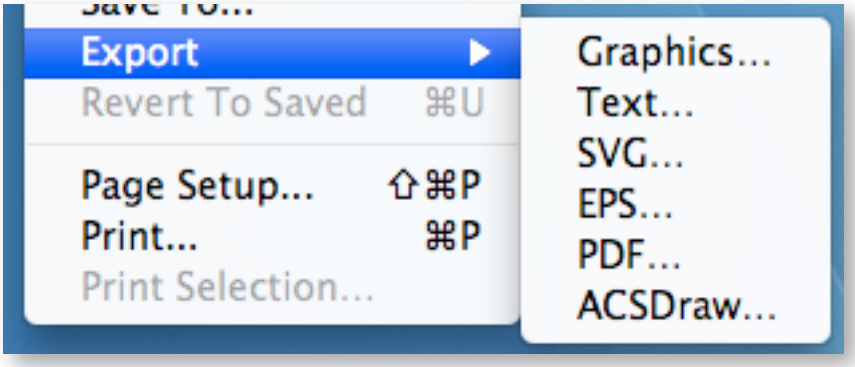
The **Print…** menu will print whichever window is frontmost, so if you want to print from the graphics window, bring that to the front; if you want to print from the main window, bring that to the front.

The Export Submenu

Manu items in the Export submenu allow you to save graphics or text. Each will prompt for a destination file to save to.

Text… is the only entry which exports text — it exports text from the main window. All the others export from the graphics window.

Graphics… exports the Graphics window as a TIFF file. This is a standard image bitmap file type which can be opened in image editors or viewers such as Photoshop or Preview. This is the only bitmap export, and exports exactly what is seen in the Graphics window.



All the other export menu items export vector graphics which can be scaled without losing resolution. Anything drawn by the turtle on the canvas is also saved as vector graphics, and this can be saved using the following menu items:

SVG — Scalable Vector Graphics — is a standard format which can be embedded in web pages. At the time of writing, current versions of Safari, Opera, and Firefox support this format.

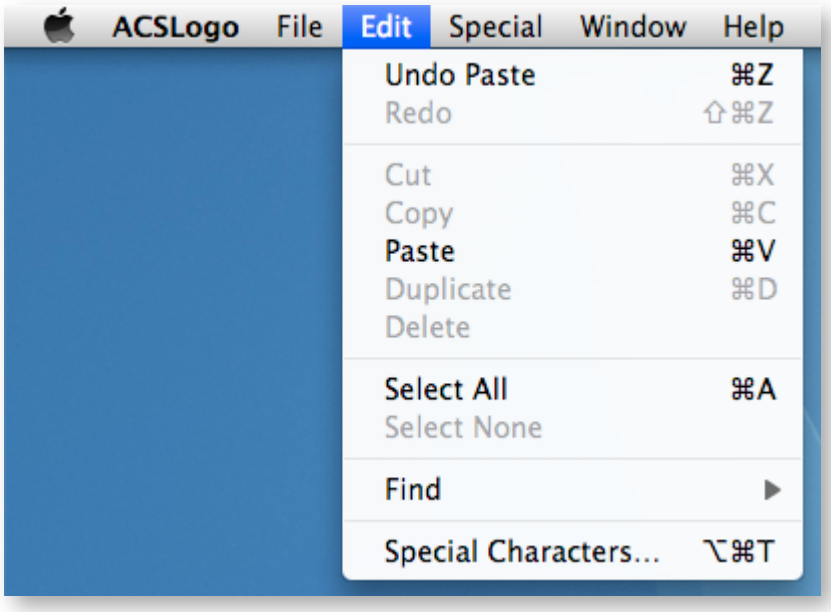
EPS — Encapsulated Postscript — outputs a file consisting of Postscript commands. Postscript is both a programming language and a page description language for printers. It's generally more practical to use PDF.

PDF has become a standard for interchange of documents. Since it can be read in OSX with its built-in software (Preview), and by the free Adobe Reader on the Mac and other platforms, it's a good choice to use.

ACSDraw is an illustration program which I have written, amongst other things, to do this documentation.

The Edit menu

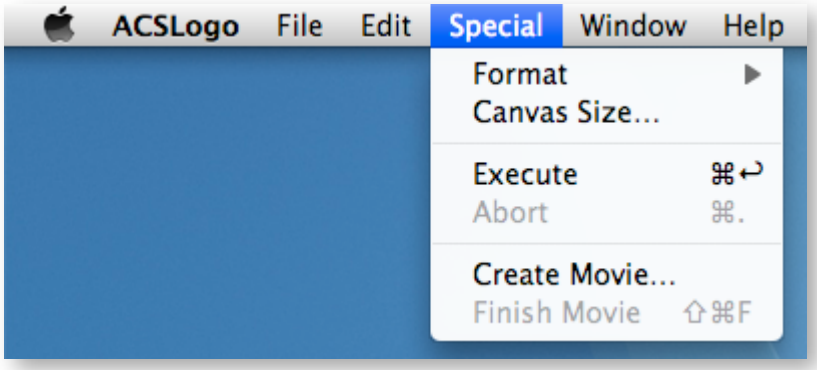
This is the standard OSX Edit menu:



It contains all the menu items found in text editing programs such as Textedit.

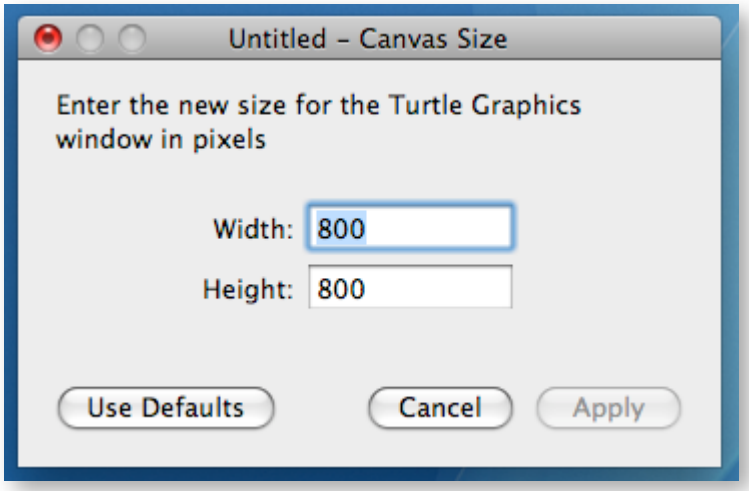
The Special menu

This is basically a group of items which don't fit anywhere else!



The **Format** submenu expands to a series of menu items which relate to text formatting. These are the same as other text-editting programs such as Textedit, so I won't go into them further here.

Canvas Size… is used to change the size of the Graphics window — a dialog is displayed allowing you to choose a new width and height for the window:



The fields are initially set to the current size. Values are in pixels.

Execute is used to execute a command. It will attempt to execute the highlighted text in the main window, or if nothing is selected, the line holding the text cursor.

Abort is the opposite to Execute. If a command is running, Abort will stop execution.

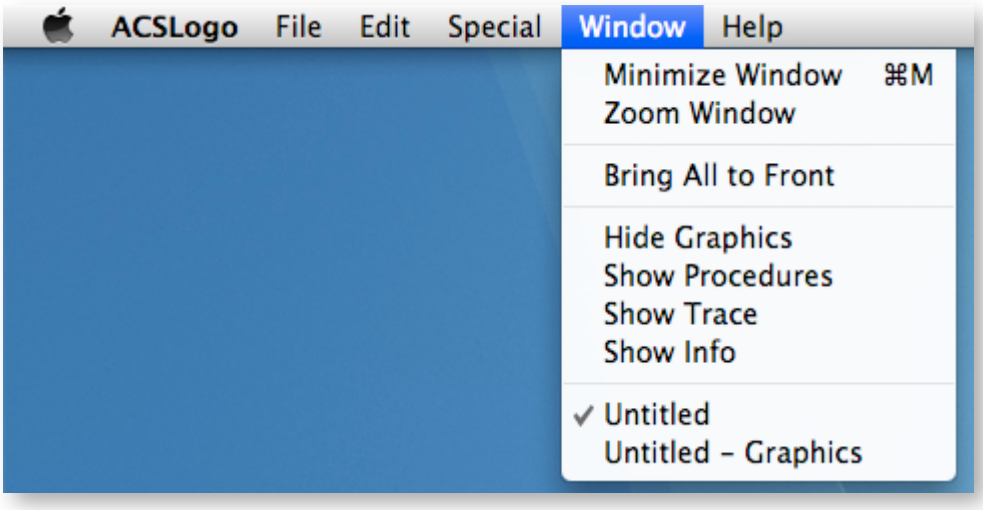
Create Movie… starts the process for exporting a Quicktime movie, first of all prompting you for somewhere to save it, then asking for the movie attributes such as compression. See the chapter Making Movies.

Finish Movie is used at the end of the movie-creating process, after capturing all the frames for the movie. It finalises the movie and closes the file. If you hold down the option (alt) key while displaying this menu, you get **Finish Movie and Open**, which opens the movie in your default application for opening Quicktime movies (usually Quicktime).

The Window Menu

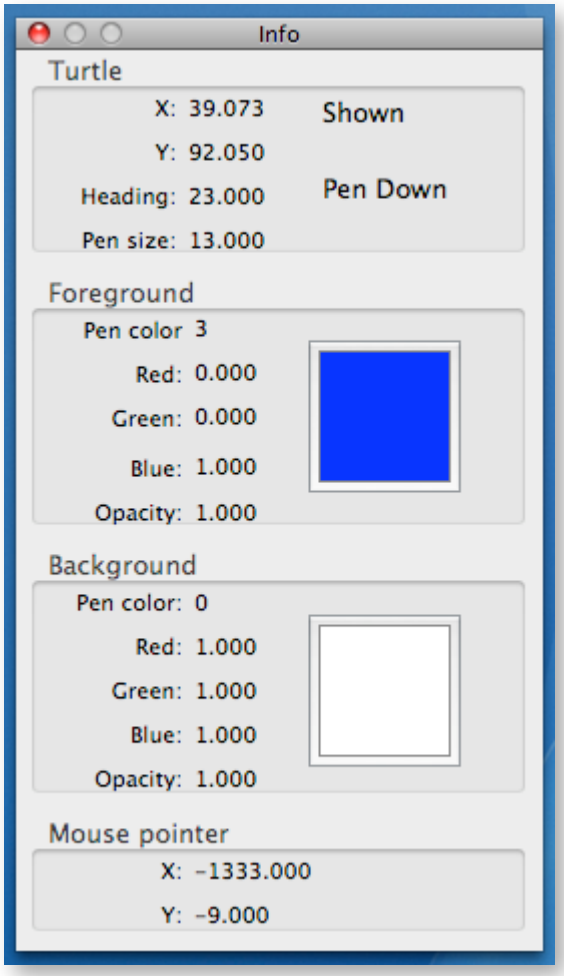
The Window menu is a fairly standard OSX Window menu. The first three entries are standard:

Minimize Window	send the window to the Dock.
Zoom Window	toggle between window sizes
Bring All to Front	put all ACSLogo windows in front of other applications' windows

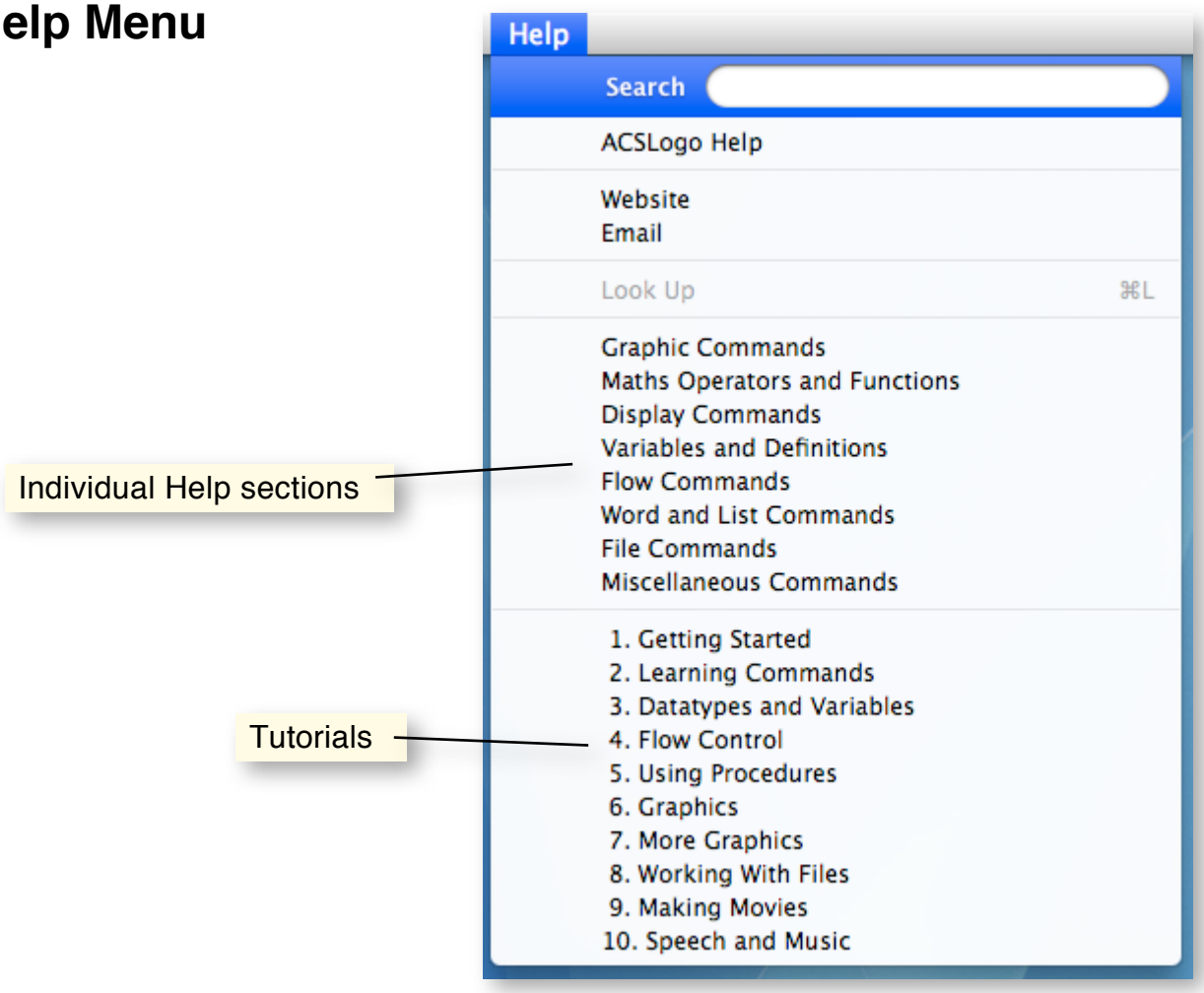


The next block of entries are to do with hiding and showing windows. They are all **Hide** or **Show** followed by a window name:

Graphics	the window containing the turtle; where all drawing takes place
Procedures	the window used to create and edit procedures. See the Procedures chapter.
Trace	Used to trace the execution of commands and procedures.
Info	Information about the state of the Turtle, current colours, etc:



The Help Menu



If you are running Leopard (OSX 10.5) or above, the first entry is the Spotlight search item. You can type text directly in here to find an item in ACSLogo Help.

The next item, **ACSLogo Help**, takes you to the help for ACSLogo in the OSX Help Viewer window.

In the next group of items, **Website** takes you to the ACSLogo website, www.alancsmith.co.uk/logo. **Email** will open a new email in your email client (such as Mail) to send me feedback.

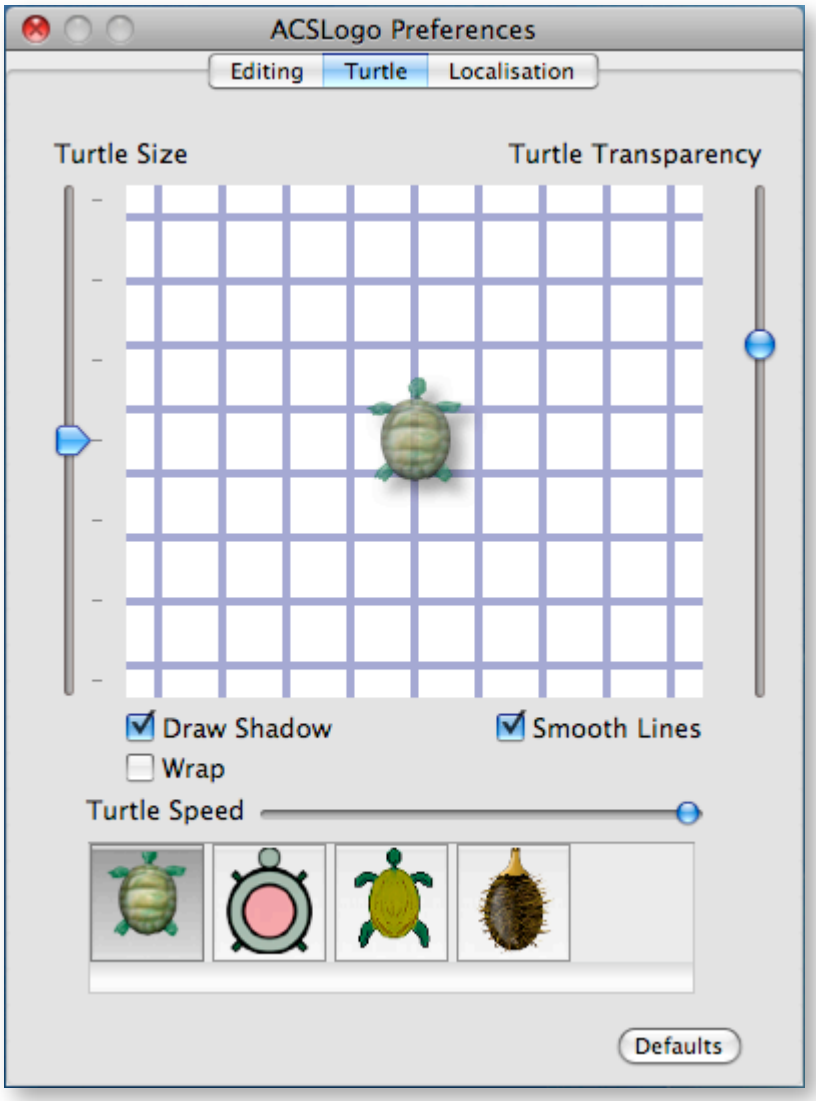
The next item, **Look Up**, is used to look up individual commands in the Help files. To use this, select a command in the main window and then choose **Look Up**. You can also hold down the command key and double-click on a command.

The next block of commands relate to individual sections in the Help files. This is quicker than choosing **ACSLogo Help** and drilling down.

The final block of entries contains the tutorials. These are files held in the **tutorials** folder which should be in the same folder as the ACSLogo application – so you can add to and remove from the folder if you like and ACSLogo will pick up the changes on restart and change the display of the menu accordingly.

Appendix B: Preferences

Choosing **Preferences…** from the ACSLogo menu, displays the Preferences window:



The preferences window has three tabs. The first one that opens is the Turtle tab.

The Turtle Tab

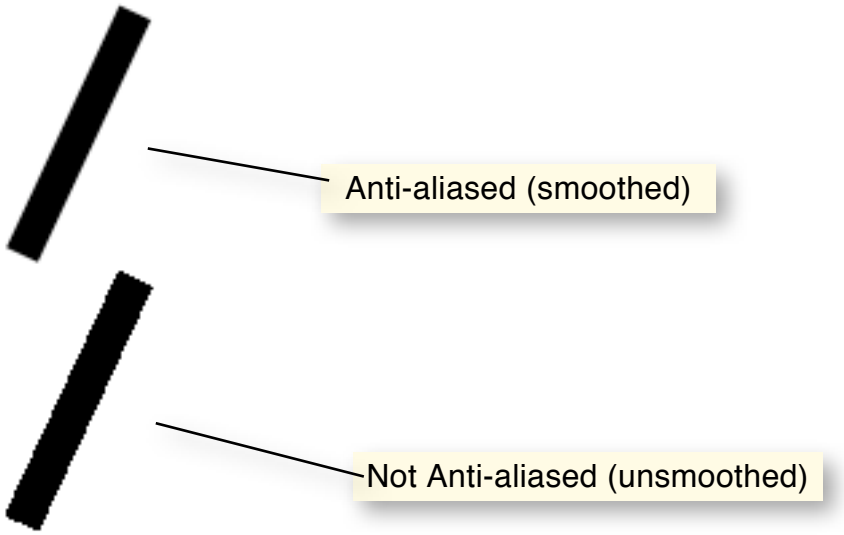
It has a large pane in the middle which shows the current appearance of the turtle. This will change as you change settings in this tab.

The **Turtle Size** slider changes the size of the turtle. Drag it upwards to increase the size of the turtle, downwards to decrease the size.

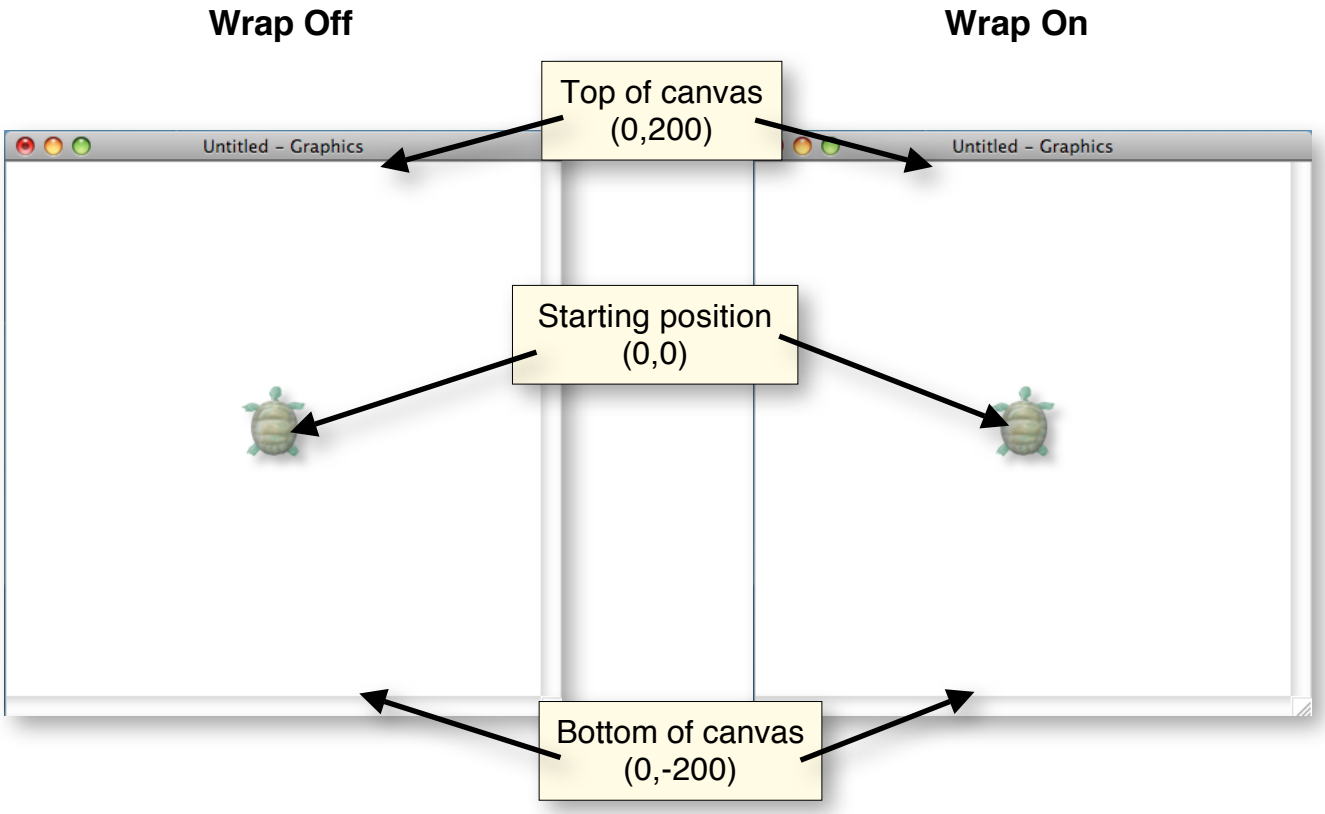
The **Turtle Transparency** slider changes the transparency of the turtle. The advantage of making the turtle more transparent (by dragging the slider down) is that you can see the drawing taking place under the Turtle, while still keeping any eye on the position and heading of the Turtle.

Draw Shadow turns on a shadow under the turtle. This is purely for appearance' sake, giving the Turtle some solidity.

Smooth Lines causes lines drawn by the turtle to be anti-aliased, or smoothed, reducing the jagged appearance. This is the technique used by OSX to make fonts appear smooth. You can see the difference with these lines:

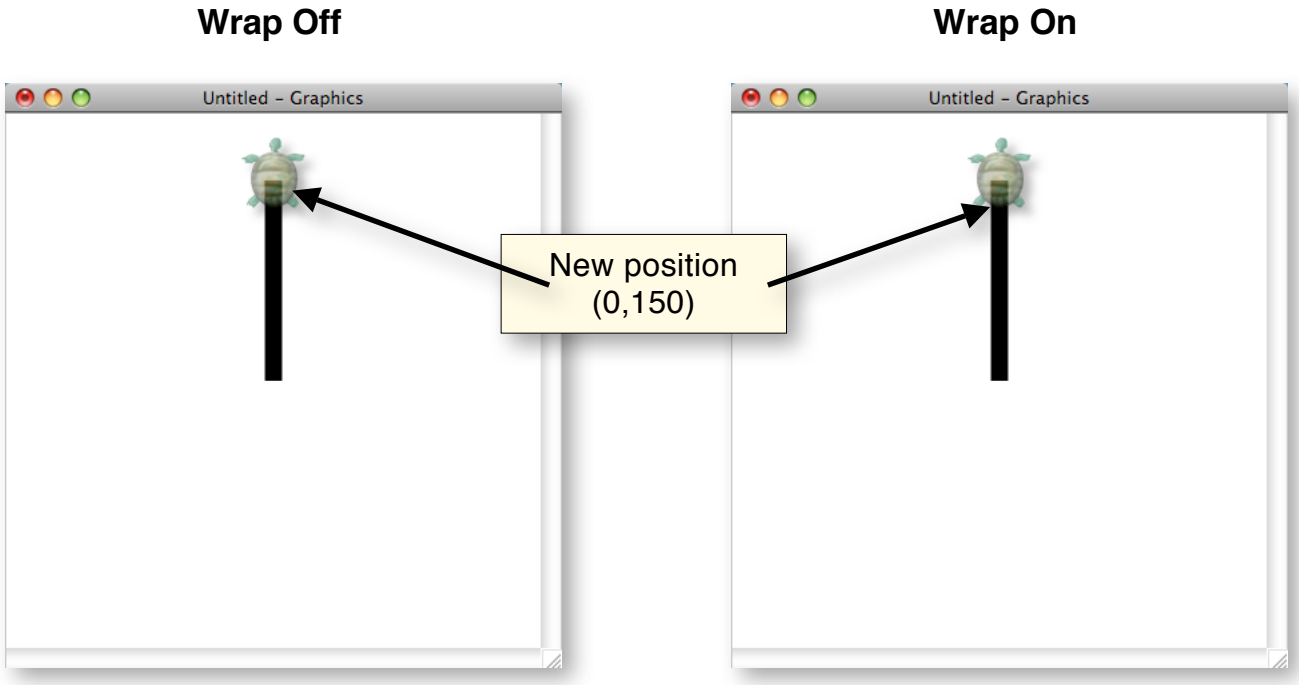


The **Wrap** checkbox determines what the turtle does when it gets to the edge of the canvas. By default, it just goes of the edge, and you can't see it anymore. If you turn on **Wrap**, the turtle comes in the other side. Let's look at an example of a window which is 400 pixels wide by 400 pixels high. The left-hand examples have **Wrap** off, the right-hand examples have it on.

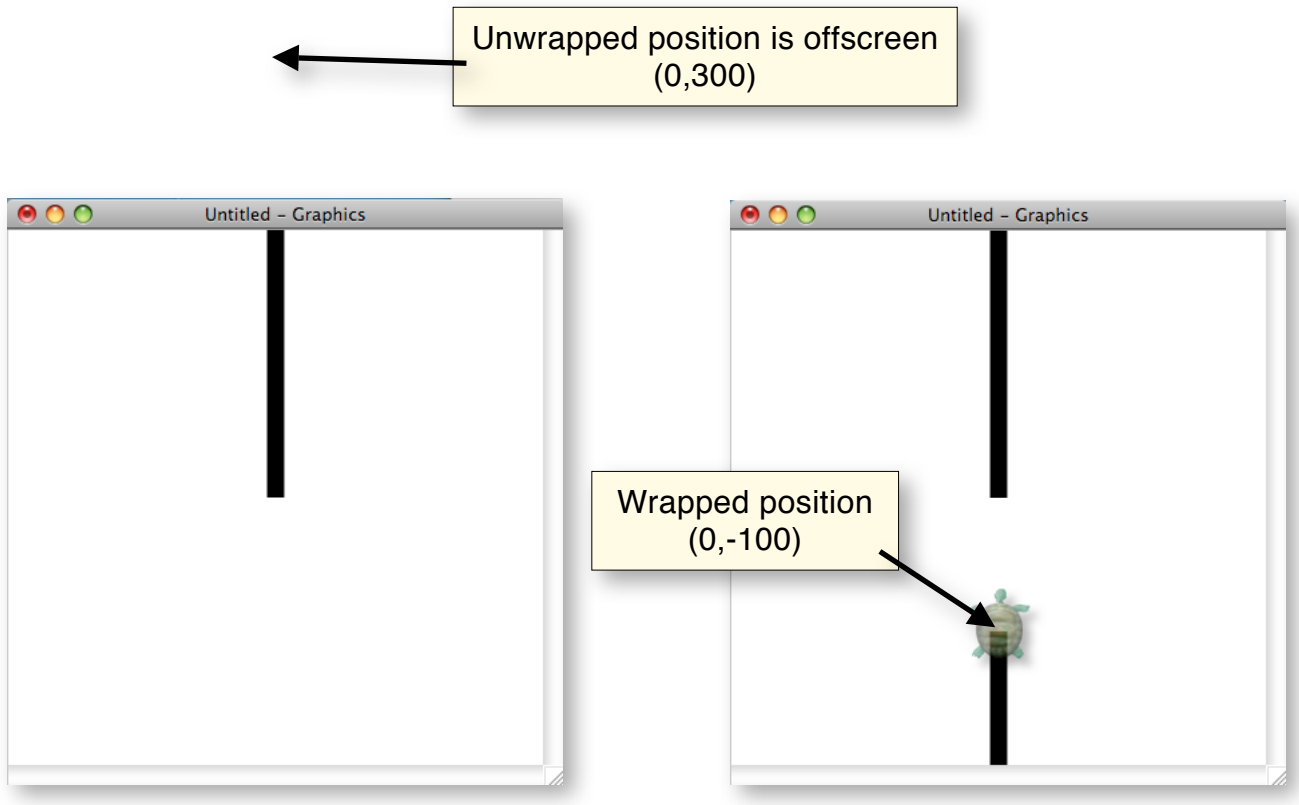


This is the starting position after a **Clearscreen** command with the turtle at position (0,0) - the x co-ordinate is zero, and the y co-ordinate is zero.

Now issue **Forward 150**. This increases the y co-ordinate by 150.



Then issue **Forward 150** again. This increases the y co-ordinate from 150 to 300.



The unwrapped Turtle has gone past the top of the canvas at y=200, so is no longer visible.

The wrapped Turtle, hitting the top of the canvas after travelling 50 pixels, comes in the bottom of the canvas. It still has 100 pixels to go (it was told to go forward 150 pixels), so draws a line from y=-200 to y=-100.

When drawing a line which is much longer than the canvas dimensions, the wrapping may occur several times. For instance, given the commands:

Right 5

Forward 1050

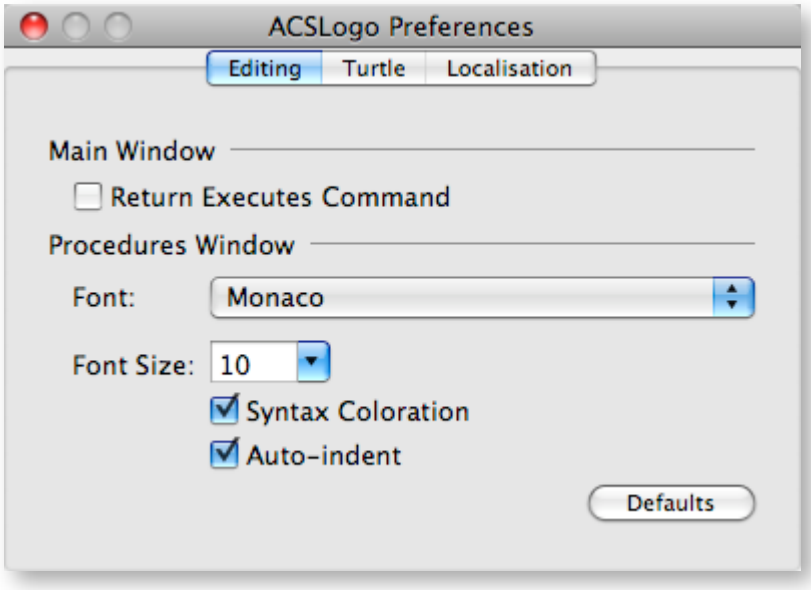
You get this result:

The **Turtle Speed** slider allows you to change the speed of the Turtle. In previous versions of the program, as the Turtle moved forward, drawing a line, the line would be drawn (taking milliseconds), and the Turtle would just be drawn at the end of the line. So, when drawing a rectangle, you would just see the Turtle drawn momentarily at each corner. This is not very good for teaching purposes, as you get no impression of the Turtle 'travelling' along it's route. The slider changes this - when the slider is at its maximum (its rightmost setting), the turtle moves as quickly as in previous versions. When you drag the slider to the left, the Turtle moves more slowly, and you can see it travel from position to position, and rotate slowly as well.

Near the bottom of the Preferences window is an array of Turtle images. Select any of these to use it as the Turtle image. You can even add your own turtle image by dragging an image into the array. The image needs to have a transparent area around the turtle – otherwise a white rectangle gets drawn around the turtle. This means you probably need to use a TIFF or a PNG image. For more details, see Appendix C: Roll your own Turtle.

At bottom right is the **Defaults** button - hit this to reset the Turtle preferences to their defaults.

The Editing Tab



Under the 'Main Window' heading is a single checkbox, **Return Executes Command**. Under the default set-up, you need to hold down the command key when you press return in order to

execute a command: pressing return on its own just inserts a line in the text. If this checkbox is checked, pressing the return key will execute the command. This makes it similar to traditional command-line versions of Logo. To insert a newline in the text, hold down the control key while pressing enter.

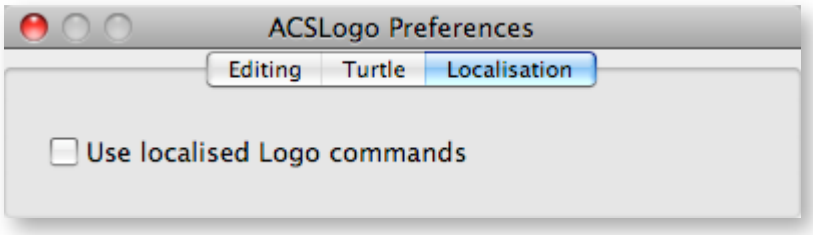
Under the **Procedures Window** heading are a number of controls which affect editing in the Procedures window. All text in the procedures window has the same font. This can be set using the **Font** and **Font Size** pulldowns.

Checking the **Syntax Coloration** checkbox causes brackets and operators to appear in different colours from the main text.

When the **Auto-indent** checkbox is checked, if you indent a line by a number of spaces and press enter to insert a newline, the next line is indented by the same amount.

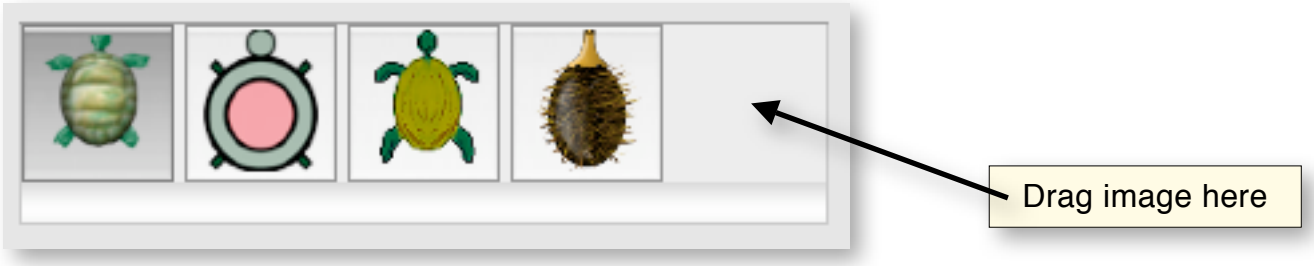
The Localisation Tab

This contains just one checkbox — **Use localised Logo commands**. If a non-English language is topmost in the Internationalisation pane of System Preferences, and a command mapping property list is available in the ACSLogo application (as with French and German), and the checkbox is checked, localised commands can be used instead of English ones.



Appendix C: Roll Your Own Turtle

The **Turtle** pane of the Preferences window allows you to define your own turtle. The array of turtles at the bottom allows you to drag in an image of your choice.



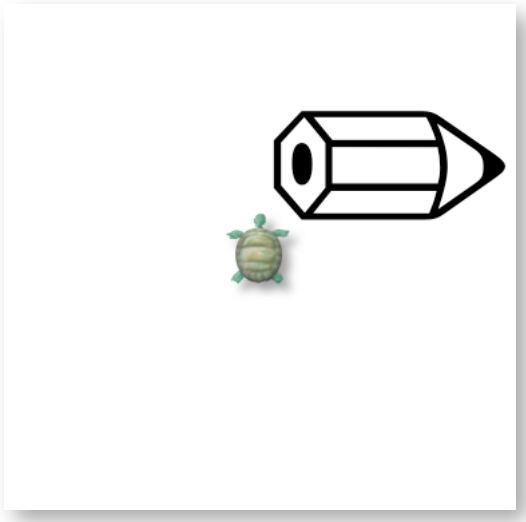
The turtle image should be square, and the 'Turtle' should be facing upwards. The area around the Turtle should be transparent rather than white so that it does not draw a white square over the background. For this reason, the image should be in a format which supports transparency such as PNG or TIFF. This can be prepared in Photoshop using layers, but if you don't have Photoshop, here's a way using a drawing done in ACSLogo itself.

Rather than creating a drawing of my own, I'll use a symbol from a symbol font.

Set your canvas size to about 400 x 400 pixels, then try these commands:

```
ClearScreen
SetFont [ZapfDingbatsITC]
SetTypeSize 200
GraphicsType [/]
```

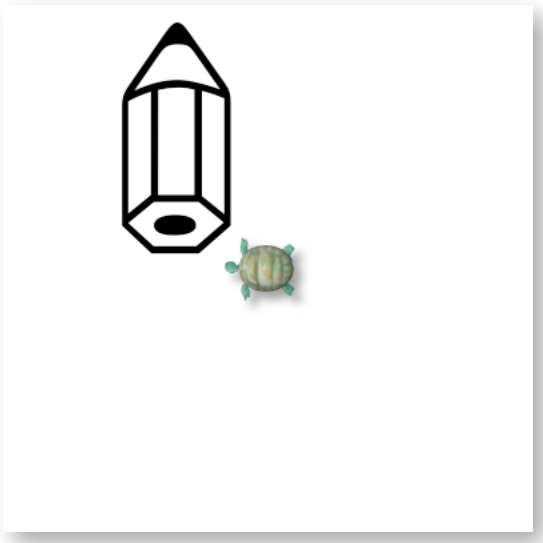
The result should look something like this:



The 'pencil' is a symbol from the Zapf Dingbats font, and is equivalent to the slash character in a normal font. I have drawn it in giant size – 200 points.

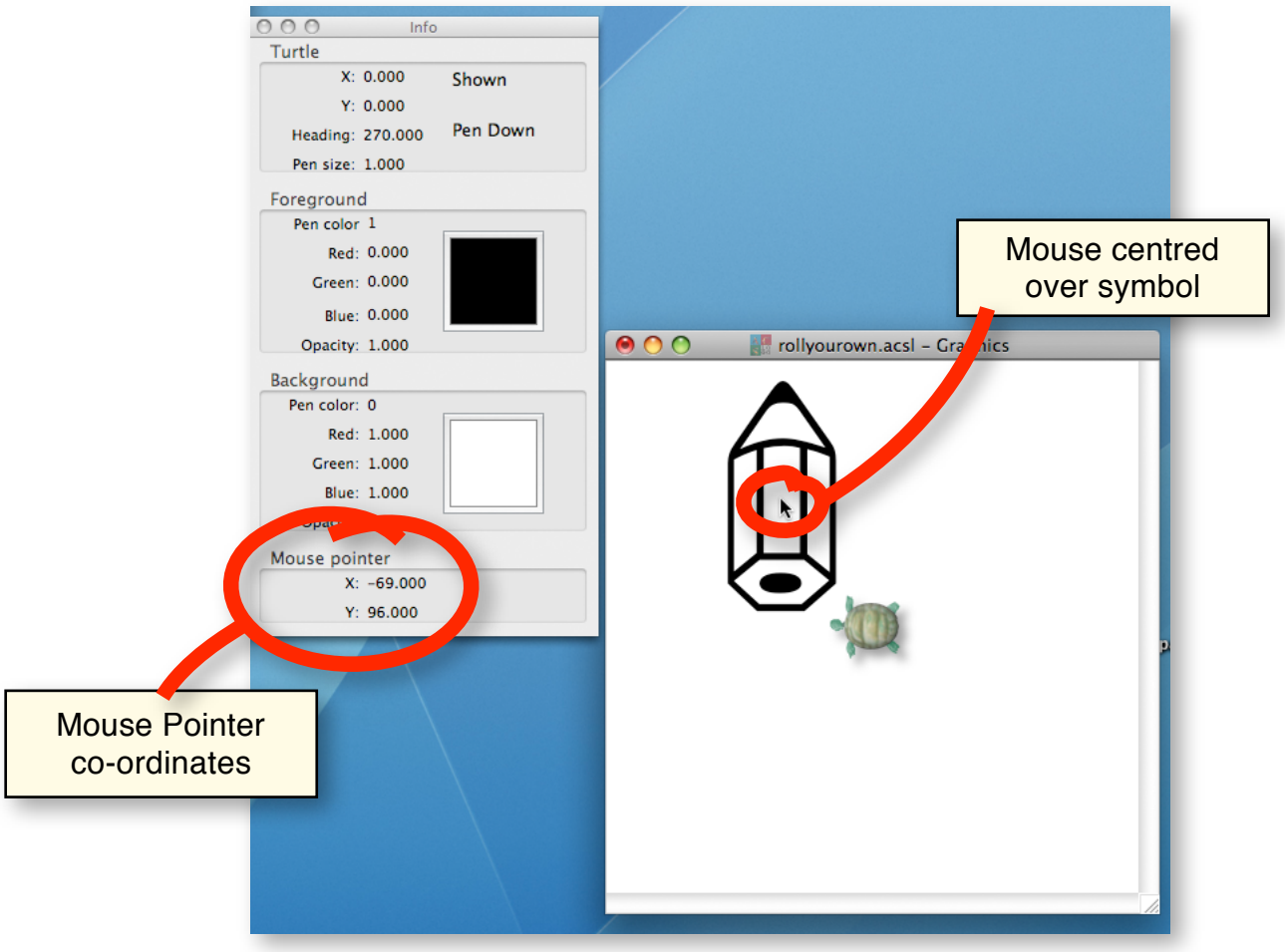
To use the symbol as a turtle, I need to centre it in the canvas and make it point upwards. It's pointing to the right at 90 degrees, so if I turn the Turtle to the left before drawing the symbol, it should have the correct orientation:

```
ClearScreen
SetFont [ZapfDingbatsITC]
SetTypeSize 200
Left 90
GraphicsType [/]
```

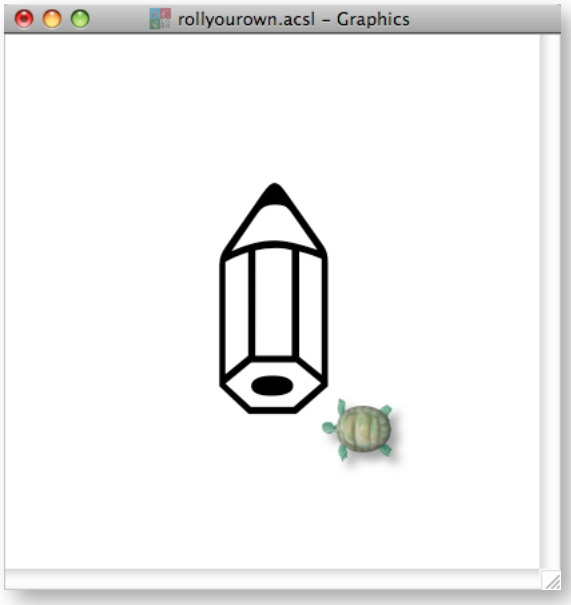


It's pointing the right way now, but it's not centred – it's offset up and to the left – so if I offset the Turtle down and to the right before drawing, it should be in the right position. I can estimate the amount to offset by using the **Info** window, and positioning the mouse pointer at the centre of the symbol.

Here, the co-ordinates are (-69, 96), so if I move the turtle to (69,-96) before drawing, the centre of the symbol should end up at (0,0).

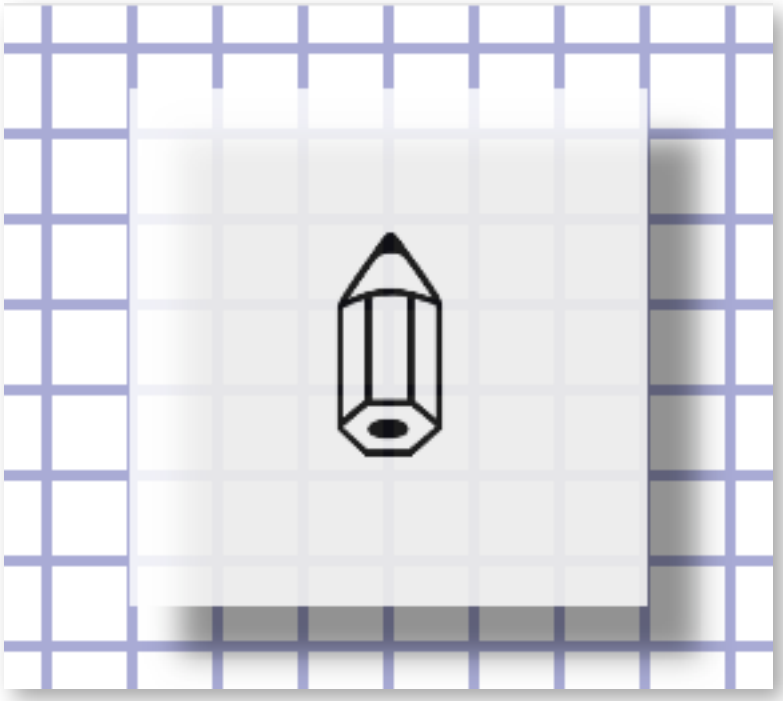


```
ClearScreen
SetFont [ZapfDingbatsITC]
SetTypeSize 200
PenUp
SetPosition [69 -96]
PenDown
Left 90
GraphicsType [/]
```



I can then export the image as a TIFF image file from the **File/Export/Graphics…** menu item.

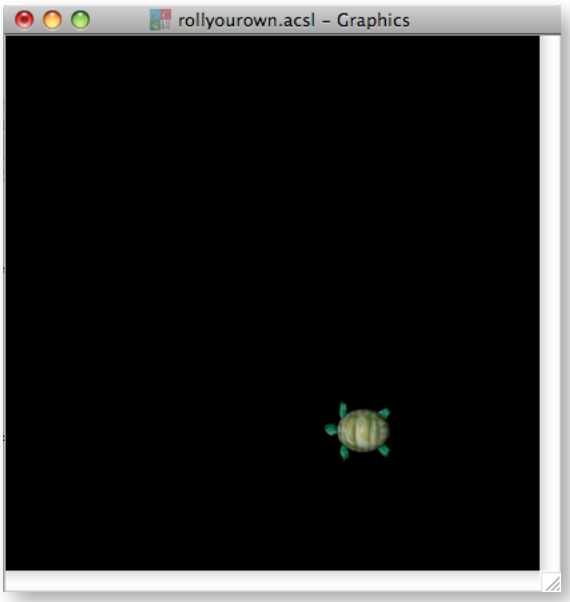
If I then drag the image into the Turtle array in the Preferences Window, I get this (I've dragged the size slider upwards to make the image bigger:



The white background from the canvas has been drawn as well, drawing over the background. This is not what we want — that area should be transparent so that only the symbol itself is drawn. Back to the drawing board, er, canvas.

So we need to draw a transparent background. This is done by setting the opacity for colour zero (the background colour) to zero. The ClearScreen command will then set the background to completely transparent:

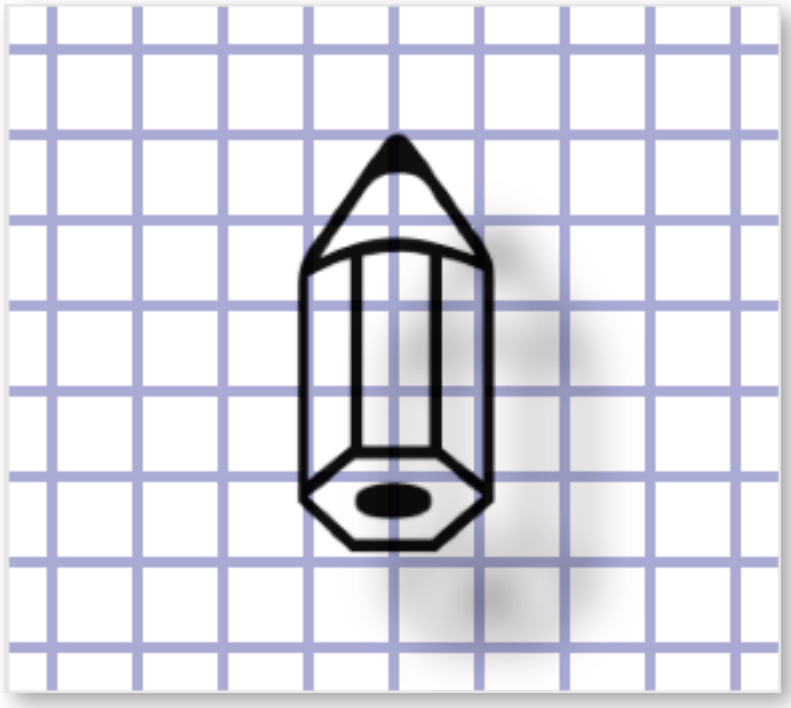
```
SetRGB 0 [0 0 0 0]
ClearScreen
SetFont [ZapfDingbatsITC]
SetTypeSize 200
PenUp
SetPosition [69 -96]
PenDown
Left 90
GraphicsType [/]
```



It looks like the background has been filled with black, but actually it's been filled with nothing. Drawing the symbol in black on this background means we can't see it, but it's still there.

Exporting the image as a Tiff again and viewing it in the finder, we can see it is there.

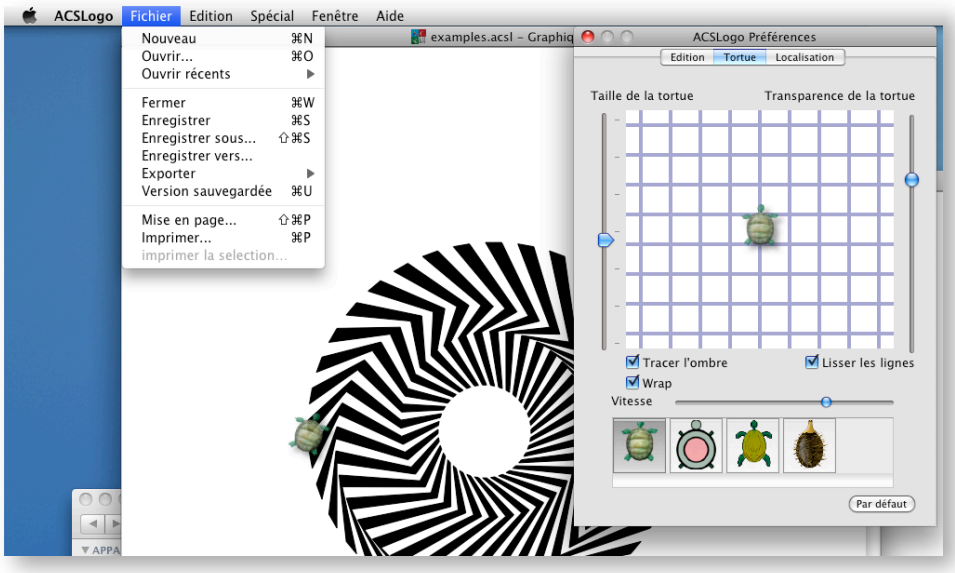
The image file needs to be dragged into the Preferences window and resized by using the size slider as required. It's then ready for use.



Appendix D: Localisation

Localisation is the customisation of a program's interface to make it usable in languages other than English.

ACSLogo, in common with most other Mac OS X applications, has always allowed a certain amount of localisation by the knowledgeable user — menus and dialogs may have their constant fields translated into a non-English language by editing the resources in the ACSLogo application bundle.



Version 1.4b added to this by allowing translated commands to be used instead of the English ones. For example, in a German translation, **Vorwart** might be used instead of **Forward**.

This means that ACSLogo can be localised for a new language without any program coding changes.

There are three parts to localisation:

The GUI	Localising the static text in menus, dialogs, etc. Also localising dynamic text such as Hide/Show menus and Undo strings.
Logo Commands	Translating the Logo commands into your chosen language.
Help and Tutorials	Localising the help files and tutorials included in the help menu.

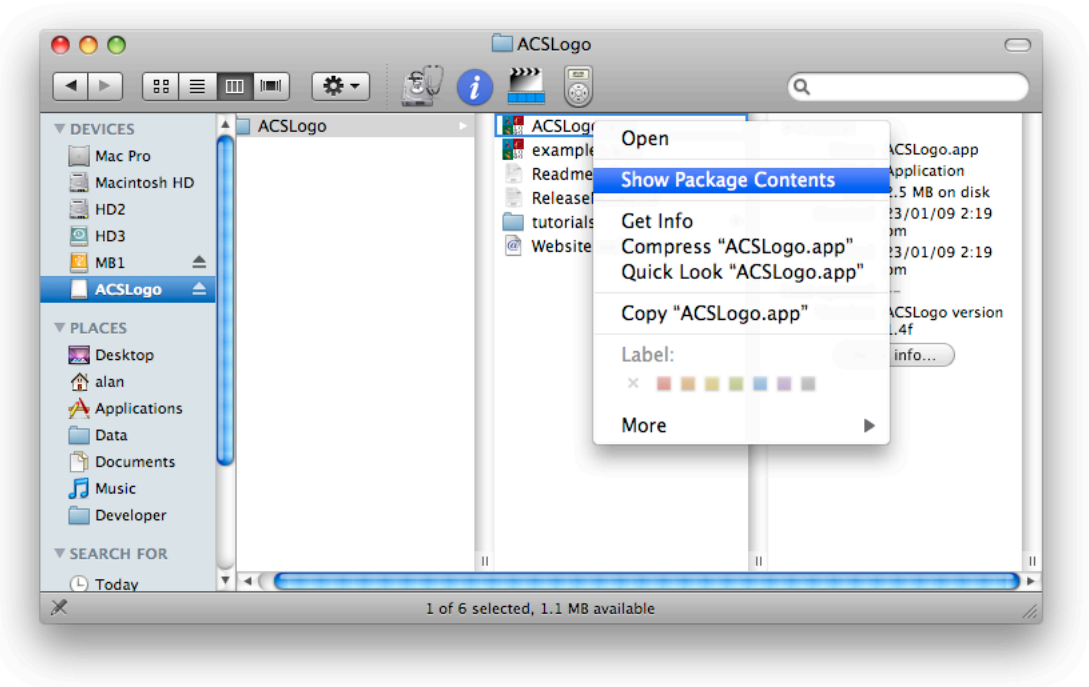
There's no compulsion to do all of these — you could for instance just do the GUI.

Prerequisites

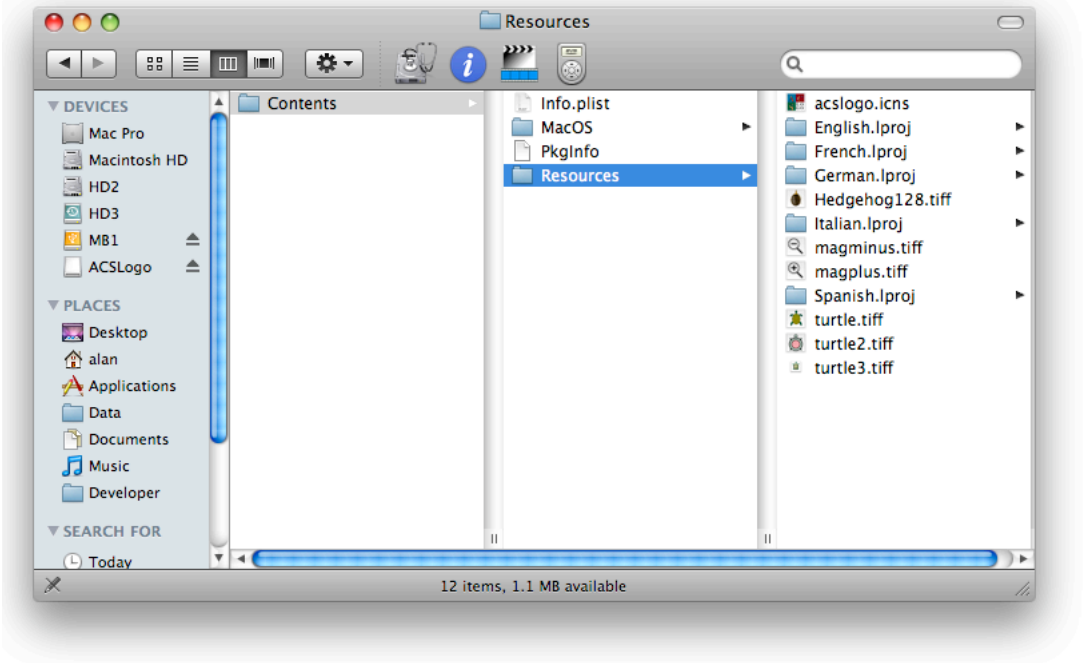
You need to have the free OS X Developer Tools installed. These come on a separate disk with the operating system, often called 'XCode Tools', but it varies with operating system release.

The Application Bundle

In the Finder, a program looks like a single file. In fact it is a ‘bundle’ — a directory of files and sub-directories including the runnable code and a number of resources. To see the structure, right-click (control-click) on the ACSLogo icon and choose ‘Show Package contents’:



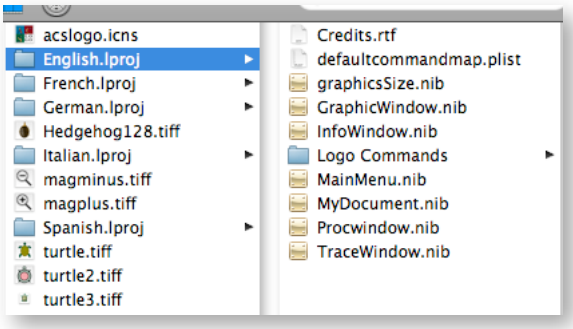
In the resulting window, choose column view if you’re not already in it. Everything we’re interested in is in the **Resources** folder. Here you can see some tiffs which are used for turtle



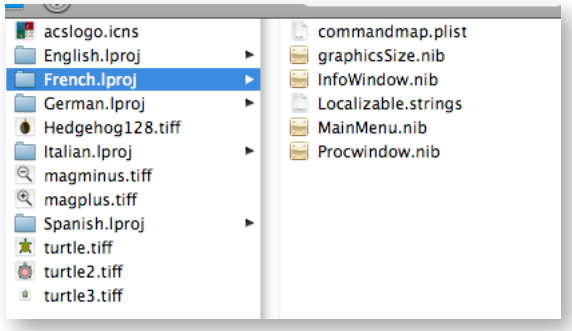
images and other images used in the program. We’re interested in the languages folders — *English.lproj, German.lproj*, etc. English.lproj is the ‘original’ folder that holds all the English language resources. You can see that some localisation has been done for French, German, Italian, and Spanish. To add another language, you would create an additional folder for that language.

Look in the English.lproj folder:

You can see the .nib files, which are the interface files representing menus, windows, etc. The *Logo commands* folder contains the help files. *defaultcommandmap.plist* is used for mapping commands.



Now have a look in the French.lproj folder:



There are fewer .nib files, because some windows did not need translating. There is no *Logo Commands* folder, so Help has not been translated. *commandmap.plist* overrides *defaultcommandmap.plist* in English.lproj. *Localizable.strings* is a list of English strings used in the program and their French equivalents.

Localisation Tasks

You first need to create a .lproj folder for your chosen language. For example, to localise in Dutch, you would create a folder called Dutch.lproj (if you look in the Resources folder for OS X programs such as Textedit or iCal, you will see the sorts of names you can use). All the localisation will be done in this folder.

The three localisation tasks are described in the following sections:

GUI Localisation

Logo Command Localisation

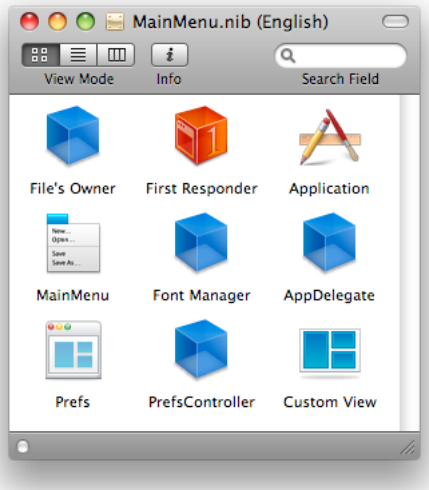
Help and Tutorial Localisation

GUI Localisation

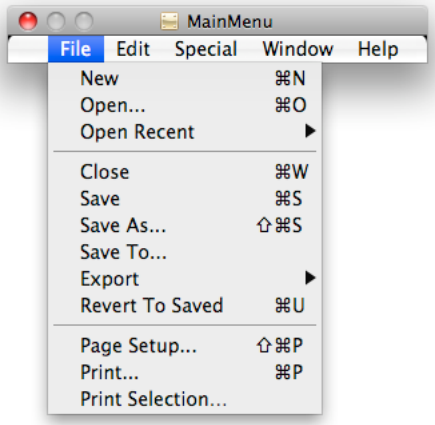
The .nib File

We'll start with the ACSLogo menus. Copy **MainMenu.nib** from English.lproj into your new .lproj folder. Double-click on the file to open it up. As long as you've installed the developer tools correctly, this'll start up the Interface Builder program. The following discussion describes Interface Builder in the Leopard release of Mac OSX. This is quite different from the Tiger version, although the principles are the same. See the Interface Builder documentation for detailed instructions on how to use the program.

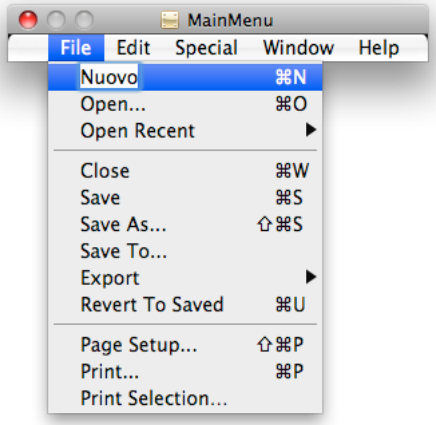
When you double-click on the .nib file, Interface Builder starts up and opens a number of windows. This is the main one:



It shows the objects in the nib file. Double-click the MainMenu instance — this will open the MainMenu object (if it's not already open) and allow you to edit it. Click on each menu to show the menu items under it.



To change the text, just double-click on it, and type in your translation:

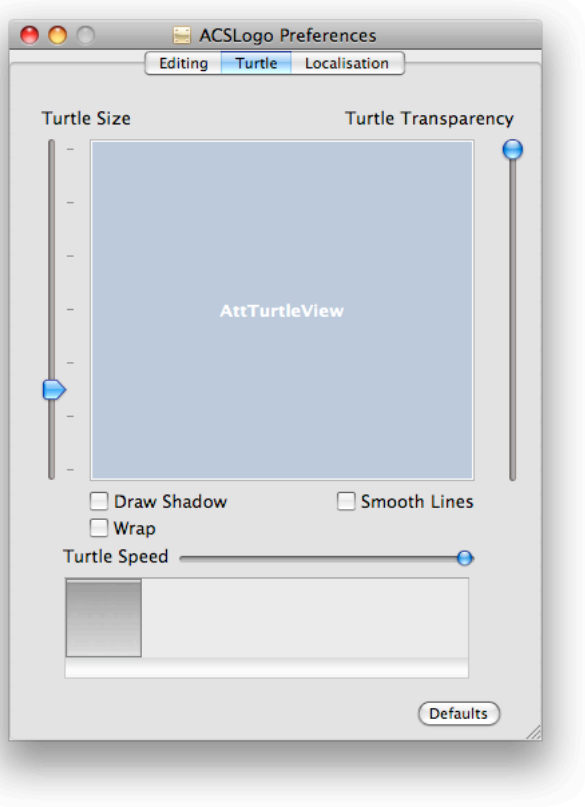


Do this to the strings in all the menus.

It's important to change the text and nothing else. Each of the menu items is connected to an action in a program object. If you duplicate or cut and paste menu items, you will mess up these connections.

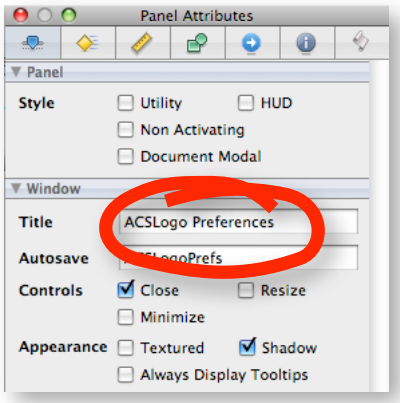
The Prefs window is also in this nib file. It's called *Prefs* in the main window. Double-click on it

to open it up.



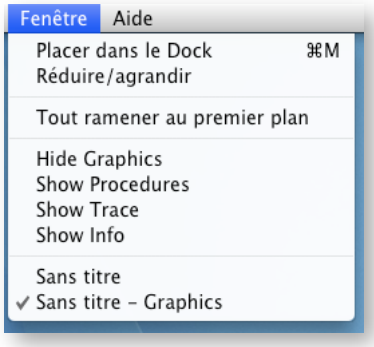
In the same way as for the menus, you can double-click on the strings to edit them. You will probably find that the size of the string has changed after translation, so you may need to resize the string or move it to keep it aligned with other objects. If you select the string by clicking on it, you can drag it or pull the handles to resize it.

To change the title of the window, select **Tools/Attributes Inspector** and key in the new title:



Save the nib file. You can now try out the program, and you should see that the menu and window strings have changed.

If you have a look in the Window menu, you'll see that some of the menu items that you translated are still in English:



This is because these items aren't picked up from the nib, they're set dynamically — the menu items have to change as windows are shown or hidden. There are a number of dynamic menu titles and other strings throughout the program. This problem is addressed using a file called *Localizable.strings*.

Localizable.strings

You can find the Localizable.strings file in any of the .lproj folders which have had some localization done. It's just a unicode text file, and you can open it in XCode. This is an excerpt from the French one:

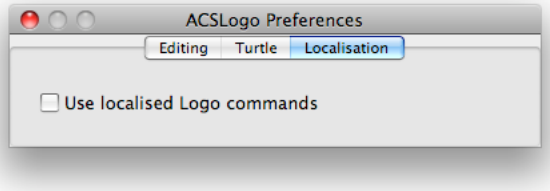
```
"Hide Procedures" = "Masquer Procédures";

"Show Procedures" = "Afficher Procédures";

"Show Graphics" = "Afficher Graphique";

"Hide Graphics" = "Masquer Graphique";
```

It consists of a number of lines of the form *English string = translated string*. So for instance, the translation for *Show Info* is *Afficher Info*. To create your own translation, copy this file into your .lproj folder and translate each string in the file. Back up the file before changing it — I've had corruptions when editing these files within XCode in the past.

Once you've saved the file, that's the interface translation finished. You can leave it there, or you can carry on localising with Logo Command Localisation.

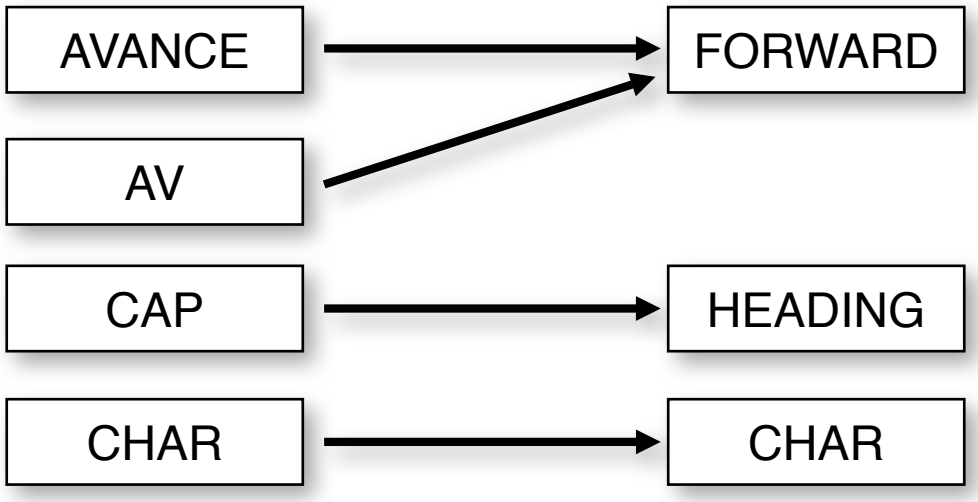
Logo Command Localisation

You can swap the set of English commands that come with ACSLogo for a set in your chosen language. This has already been done for French and German. When it's been done, the user has to choose localised commands using the Localisation pane in preferences:

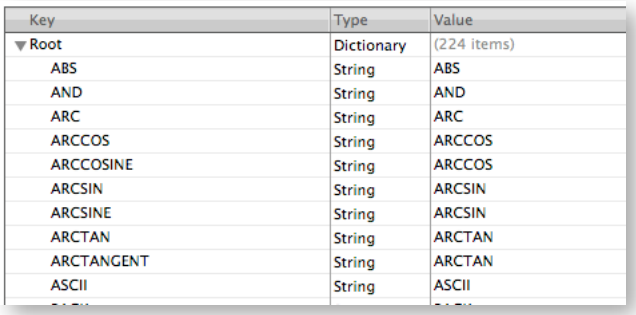


Checking the switch will turn on localised commands if they exist (of course, the preferences panel will appear in the current language).

You can see the full list of commands that need to be translated and their equivalents in French and German in the command table. Localising the commands involves creating a list of commands translated from the English ones. This creates a command mapping:



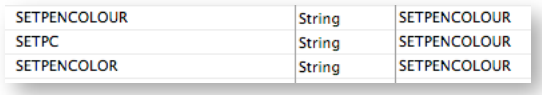
So the French command **Avance** and its short version **AV** need to translate to **Forward**, **Cap** translates to **Heading**, etc. In fact, this kind of mapping even occurs for English — a command like **SetPenColour** may have several variants, such as a short form (**SetPC**) and an alternative spelling (**SetPenColor**). These three forms map to the core ACSLogo command, **SetPenColour**. You can see this mapping in the *defaultcommandmap.plist* file which is in the *English.lproj* folder. Double-click it to open it – it should open in the Property List Editor.



Key	Type	Value
▼ Root	Dictionary	(224 items)
ABS	String	ABS
AND	String	AND
ARC	String	ARC
ARCCOS	String	ARCCOS
ARCCOSINE	String	ARCCOS
ARCSIN	String	ARCSIN
ARCSINE	String	ARCSIN
ARCTAN	String	ARCTAN
ARCTANGENT	String	ARCTAN
ASCII	String	ASCII

The Root of the file is a dictionary – a list of keys and values. the keys are the commands which the user is allowed to type in ACSLogo; the values are the core ACSLogo commands. So the keys can be anything — you could add a key (and hence a command) called Piffle — but its value would have to be from the finite set of core ACSLogo commands. You can see from the start of the dictionary that it is mostly commands mapping to an identical value — ABS to ABS, AND to AND, etc., but down a few entries you can see that both ARCCOS and ARCCOSINE map to ARCCOS – so the user can use both.

Further down the list, you can see that SETPENCOLOUR, SETPC, and SETPENCOLOR map to SETPENCOLOUR.

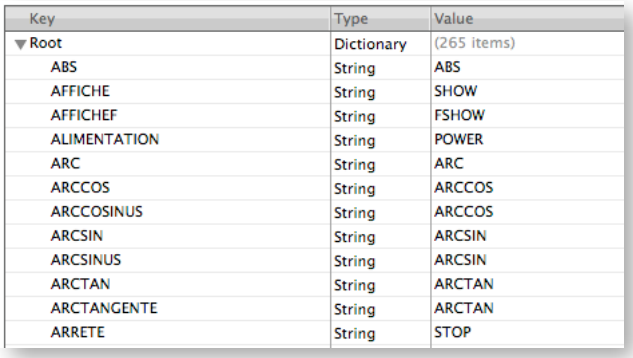


SETPENCOLOUR	String	SETPENCOLOUR
SETPC	String	SETPENCOLOUR
SETPENCOLOR	String	SETPENCOLOUR

In general, you will not be changing this list for localisation, because it relates to mapping English commands. You need to change a file called *commandmap.plist* found in the localised .lproj folders. Got the French.lproj and open the one found there.

You can see that it's exactly the same format as *defaultcommandmap.plist*, the difference

being that the keys are French commands. The English commands on the right are the same core



Key	Type	Value
▼ Root	Dictionary	(265 items)
ABS	String	ABS
AFFICHE	String	SHOW
AFFICHEF	String	FSHOW
ALIMENTATION	String	POWER
ARC	String	ARC
ARCCOS	String	ARCCOS
ARCCOSINUS	String	ARCCOS
ARCSIN	String	ARCSIN
ARCSINUS	String	ARCSIN
ARCTAN	String	ARCTAN
ARCTANGENTE	String	ARCTAN
ARRETE	String	STOP

commands as in *defaultcommandmap.plist*.

To create your own *commandmap.plist* file, copy *defaultcommandmap.plist* into your .lproj folder and rename it. Edit the file and overtype the keys with the translations into your own language. You can add entries, but the Value on the right must be one of the core ACSLogo commands from *defaultcommandmap.plist*. You can remove entries, but if nothing maps to one of the core commands, it will be inaccessible to the user. Once you've saved the file, turned on localisation in the ACSLogo preferences, and restarted ACSLogo, your new commands are ready for use.

When you restart the program, check in the Console utility (found in Applications/Utilities) to see if there are any error messages.

```
31/01/2009 18:48:25 ACSLogo[49747]  ACSLogo: mapped command CLEARSCREEV not found
31/01/2009 18:48:25 ACSLogo[49747]  ACSLogo: mapped command FONTFACES not found
31/01/2009 18:48:25 ACSLogo[49747]  ACSLogo: command EOFP not mapped
31/01/2009 18:48:25 ACSLogo[49747]  ACSLogo: command PATHLENGTH not mapped
```

The first two messages show that the values used in the command map are not core commands, so the mapping for just those commands has failed. The last two messages show that the two core commands have not been mapped to at all, so entries should be added.

Command Table

This is the list of English Commands with French and German equivalents. Core English commands are in bold.

English	French	German
ABS	ABS	BETRAG, ABS
AND	ET	ALLE
ARC	ARC	ARC
ARCCOS, ARCCOSINE	ARCCOS, ARCCOSINUS	ARCCOS, ARCCOSINE
ARCSIN, ARCSINE	ARCSIN, ARCSINUS	ARCSIN, ARCSINUS
ARCTAN, ARCTANGENT	ARCTANGENTE, ARCTAN	ARCTAN, ARCTANGENS
ASCII	ASCII	ASCII
BACK	RE, RECULE	RW, RÜCKWÄRTS
BACKGROUND, BG	FOND, COULEURFOND, CF	HINTERGRUND, HG
BUTFIRST, BF	SP, SAUFPREMIER	OE, OHNEERSTES
BUTLAST, BL	SD, SAUFDERNIER	OL, OHNELETZTES
BUTTON?, **BUTTONP**	BOUTON?	SCHALTFLÄCHE
CATCH	ATTRAPE	FANGEAB
CD	CD	CD
CHAR	CHAR, CAR	ZEICHEN
CLEAN	NETTOIE	LÖSCHE, NB, LB, NEUBILD
CLEARSCREEN, CS	VE, EFFACEECRAN, VIDEECRAN, EE, VIDE_ECRAN	LÖSCHESCHIRM
CLOSEREADFILE	FERMELECTURE	LESEDATEISCHLIEßEN, LESEDATEISCHLIESSEN
CLOSEWRITEFILE	FERMEECRITURE	SCHREIBDATEISCHLIESSEN, SCHREIBDATEISCHLIEßEN
COSINE, COS	COSINUS, COS	COSINUS, COS
COUNT	COMPTE	ANZAHL, LÄNGE
CURRENTPATH	TRACECOURANT	AKTUELLERPFAD
DATE	DATE	DATUM
DEFINE	DEFINIS, DEF	DEF
DEFINE?, **DEFINEP**	DEFP, DEFINIS?, DEFINISP	DEF?
DIFFERENCE	DIFFERENCE, DIFF	DIFFERENZ
DIR	DIR, CATALOG, CAT	INHALT
DOT	POINT	PUNKT
EDIT	EDITE, ED	BEARBEITEN
EMPTY?, **EMPTYP**	VIDE?, VIDEP	LEER?
END	FIN	ENDE
EOF?, **EOFP**	EOF?	EOF?, DATEIENDE?
EQUALP, EQUAL?	EGALE?, EGALEP, EGALP, EGAL	GLEICH?
EXP	EXP	EXP
EXPORTEPS	EXPORTEREPS	EXPORTEPS
EXPORTPDF	EXPORTERPDF	EXPORTPDF
EXPORTTIFF	EXPORTERTIFF	EXPORTTIFF
FILL	PEINS, REMPLIS	FÜLLE
FILLCURRENTPATH	REMPLITTRACECOURANT	FÜLLEAKTUELLENPFAD
FILLIN	REMPLISDEDANS	FÜLLFARBE
FILLPATH	PEINSCHEMIN	FÜLLEPFAD
FIRST	PREMIER	ERSTES, ER
FIRSTPUT, FPUT	METSPREMIER, METSP	ME, MITERSTEM
FONTFACE, FONT	ASPECTPOLICE, POLICE	SCHRIFTART, SCHRIFT
FONTFAMILIES	FAMILLESPOLICE	SCHRIFTFAMILIEN
FONTFAMILY	FAMILLEPOLICE	SCHRIFTFAMILIE
FONTS, FONTFACES	POLICES, ASPECTSPOLICES	SCHRIFTEN, SCHRIFTARTEN
FONTTRAITS	STYLE	SCHRIFTEIGENSCHAFTEN
FORWARD, FD	AVANCE, AV	VORWÄRTS, VW
FPRINT	ECRITFICHIER, ECRITF	DDRUCKE
FREADCHAR	LITCARF, LITCARACTEREFICHIER	DLIESBUCHSTABE
FREADCHARS	LITCARACTERESFICHIER, LITCARSF	DLIESBUCHSTABEN
FREADLIST	LITLISTEF, LITLISTEFICHIER	DLIESLISTE
FREADWORD	LITMOTFICHIER, LITMOTF	DLIESWORT
FSHOW	AFFICHEF	DZEIGE
FTYPE	TAPEFICHIER, TAPEF	DSCHREIBE

GETMOUSECHANGE		ERKENNEMAUSVERÄNDERUNG, EMAUSV
GETMOUSECLICK		ERKENNEMAUSKLICK, EMAUSK
GETMOUSEMOVED		ERKENNEMAUSBEWEGUNG, EMAUSB
GETPROP	RPROP	NIMMEG, NIMMEIGENSCHAFT
GO	VA	GEHEZU
GRAPHICSTYPE, GRTYPE	TYPEGR, TYPEGRAPHIQUE, ECRISGRAPHIQUE, ECRISGR	GRAFIKTYP
HEADING	CAP	KURS, WINKEL
HIDETURTLE, HT	CACHETORTUE, CT	VERSTECKIGEL, VI
HOME	ORIGINE	MITTE
IF	SI	WENN
INSTRUMENTS	INSTRUMENTS	INSTRUMENTE
INTEGER, INT	ENTIER, ENT	GANZZAHL
ITEM	ELEMENT, ITEM	ELEMENT, ELT
LABEL	LABEL, ETIQUETTE	IGELTEXT, IT
LAST	DERNIER	LETZTES, LZ, LE
LASTPUT, LPUT	METSD, MD, METSDERNIER	ML, MITLETZTEM
LEFT, LT	GAUCHE, GA, TG, TOURNEGAUCHE	LINKS, LI
LIST	LISTE	LISTE
LIST?, **LISTP**	LISTE?, LISTEP	LISTE?
LOCAL	LOCALE	LOKAL
LOG, LN	LOG, LN	LOG, LN
LOG10	LOG10	LOG10
LOWERCASE	MINUSC, MINUSCULE	KLEINBUCHSTABEN
MAKE	RELIE	SETZE
MEMBER?, **MEMBERP**	MEMBRE?, MEMBREP	MITGLIED?
MOUSE		MAUS
NAME?, **NAMEP**	NOM?	NAME?
NOT	NON	NICHT
NUMBERP, NUMBER?	NOMBREP, NOMBRE?	ZAHL?
OPENAPPEND	OUVREAJOUT	ÖFFNEZUMANHÄNGEN
OPENREAD	OUVRELECTURE	ÖFFNEZUMLESEN
OPENTEXTAPPEND	OUVRETEXTEAJOUT	ÖFFNETEXTANHÄNGEN
OPENTEXTREAD	OUVRETEXTELECTURE	ÖFFNETEXTLESEN
OPENTEXTWRITE	OUVRETEXTEECRITURE	ÖFFNETEXTSCHREIBEN
OPENWRITE	OUVREECRITURE	ÖFFNEZUMSCHREIBEN
OR	OU	EINES
OUTPUT, OP	RET, SORTIE, RETOURNE, SOR	RÜCKGABE
PATHBOUNDS	LIMITESTRACE	PFADGRENZE
PATHLENGTH		PFADLÄNGE
PEN	CRAYON, PLUME	STIFT
PENCOLOUR, PENCOLOR, PC	CPLUME, COULEURCRAYON, CCRAYON, CC, COULEURPLUME	FARBEZ
PENDOWN, PD	BC, BAISSEPLUME, BAISSECRAYON	SA, STIFTAB
PENUP, PU	LEVECRAYON, LC	SH, STIFTHOCH
PENWIDTH	TAILLECRAYON, TAILLEPLUME	STIFTBREITE
PI	PI	PI
PLAY	JOUERINSTRUMENT, JOUER	SPIELE
POSITION, POS	POSITION, POS	IGELORT, ORT
POWER	PUISSANCE	POTENZ
PRINT, PR	EC, ECRIS	DRUCKEZEILE, DR, DZ, DRUCKE
PRODUCT	PRODUIT	PRODUKT
PROPLIST, PLIST	LPROP	EIGENSCHAFTENLISTE
PUTPROP, PPROP	RAPE	MITEG, GIBEG, MITEIGENSCHAFT, GIBEIGENSCHAFT
PWD	REPERTOIRE, REP	ORDNER
QUOTIENT	QUOTIENT	QUOTIENT
RANDOM	HASARD	ZUFALL, ZZ
READCHAR, RC	LITCAR, LITCARACTERE	LIESBUCHSTABE, LBU
READCHARS, RCS	LITCARACTERES, LITCARS	LBUN, LIESBUCHSTABEN
READLIST, RL	LITLISTE	LIESLISTE, LL
READWORD, RW	LITMOT	LW, LIESWORT
REMAINDER	RESTE	REST
REMPROP	EFPROP	MERKEEIGENSCHAFT
REPEAT	REPETE	WH, WIEDERHOLE

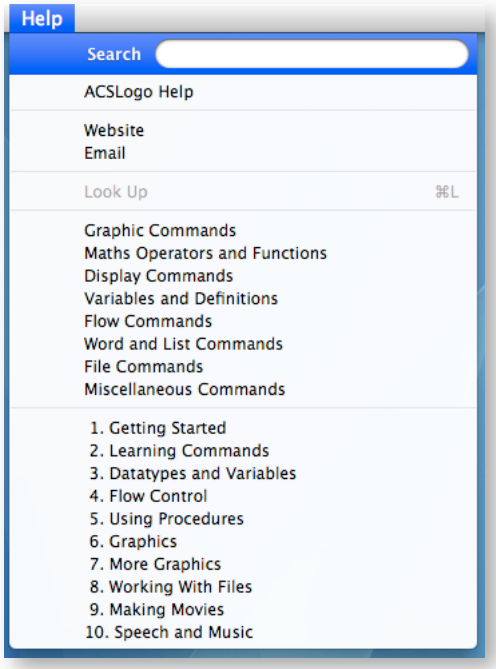
REVERSEPATH	INVERSETRACE	PFADUMKEHREN
RGB	RVB	RGB
RIGHT, RT	DR, DROITE	RECHTS, RE
ROUND	ARRONDI	RUNDE
RUN	EXEC, EXECUTE	TUE
SAY	PRONONCE	SAGE
SENTENCE, SE	PHRASE, PH	SATZ
SETBACKGROUND, SETBG	FIXECOULEURFOND, FCF	AUFHINTERGRUND
SETCANVASSIZE	CHANGETAILLECANEVAS	
SETFONTFACE, SETFONT	FIXEASPECTPOLICE, FIXEPOLICE	AUFSCHRIFTART, AUFSCHRIFT
SETFONTFAMILY	FIXEFAMILLEPOLICE	AUFSCHRIFTFAMILIE
SETFONTTRAITS	FIXESTYLE	AUFSCHRIFTEIGENSCHAFTEN
SETHEADING	FC, FIXECAP	AUFKURS, AK
SETLINECAP		LEINWANDGRÖSZESETZEN
SETLINEDASH	POINTILLES	AUFSTRICHLINIE
SETPEN	FIXECRAYON, FIXEPLUME	AUFSTIFT
SETPENCOLOUR, SETPENCOLOR, SETPC	FCC, FIXECOULEURCRAYON	FARBE
SETPENWIDTH	FTP, FIXETAILLEPLUME, FIXETAILLECRAYON, FTC	AUFSTIFTBREITE
SETPOSITION, SETPOS	FIXEPOS, FIXEPOSITION, FPOS, FPOSITION	AUF, AUFXY
SETRGB	FIXERVB	AUGRGB
SETSHADOW	FIXEOMBRE	AUFSCHATTEN
SETTYPESIZE	FIXETAILLE	AUFSCHRIFTGRÖßE
SETVOICE	FIXEVOICE	AUFSTIMME
SETX	FIXEX	AUFX
SETY	FIXEY	AUFY
SHOW	MONTRE, AFFICHE	ZEIGE, DZL, GIB
SHOWN?, **SHOWNP**	ESTAFFICHE?, VISIBLE	SICHTBAR?
SHOWTURTLE, ST	MT, MONTRETORTUE	ZGI, ZI, ZEIGIGEL
SINE, SIN	SINUS, SIN	SINUS
SNAP	CAPTUREIMAGE, CAPTURE	KNIPSE
SQRT	RACINE	QUADRATWURZEL, QW, WURZEL
STOP	ARRETE	HALT, STOPP
STROKECURRENTPATH	DESSINETRACECOURANT	AKTUELLERPFADUMRISS
STROKEPATH	DESSINETRACE	PFADUMRISS
SUM	SOMME	SUMME
TANGENT, TAN	TAN, TANGENTE	TAN, TANGENS
TEXT	TEXTE	TEXT
TEXTBOX	ZONETEXTE	TEXTBOX
THING	CHOSE	WERT
THROW	RENVOIE	WIRFZURÜCK
TIME	HEURE	ZEIT
TOWARDS	VERS	RICHTUNG, RI
TYPE	TYPE	SCHREIBE
UPPERCASE	MAJUSCULE	GROßBUCHSTABEN
VOICE	VOIX	STIMME
VOICES	LISTEVOIX	STIMMEN
WAIT	ATTENDS	WARTE
WAITFORSPEECH	ATTENDSPAROLE	WARTEAUFSPRACHE
WORD	MOT	WORT
WORD?, **WORDP**	MOTP, MOT?	WORT?
XPOS	POSX, XPOS	XORT
YPOS	YPOS, POSY	YORT

Help and Tutorial Localisation

There are a number of different bits of documentation that come with ACSLogo (including this user guide). Two of them are integrated with the program — Tutorials and Help. To localise them, you have to have some knowledge of how they fit into the program.

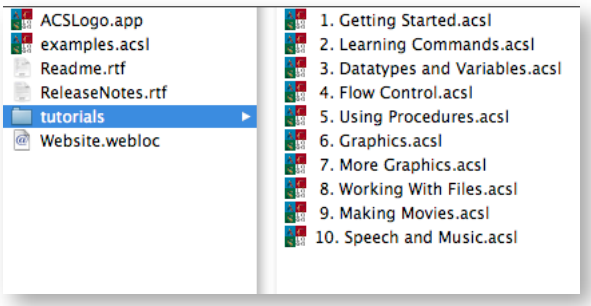
Tutorials

When you start up ACSLogo, you can see the tutorials at the bottom of the Help menu:



It's easy to substitute your own tutorials.

When you download ACSLogo, you get an ACSLogo folder which contains the program itself and some other things. One of the other things is a tutorials folder:

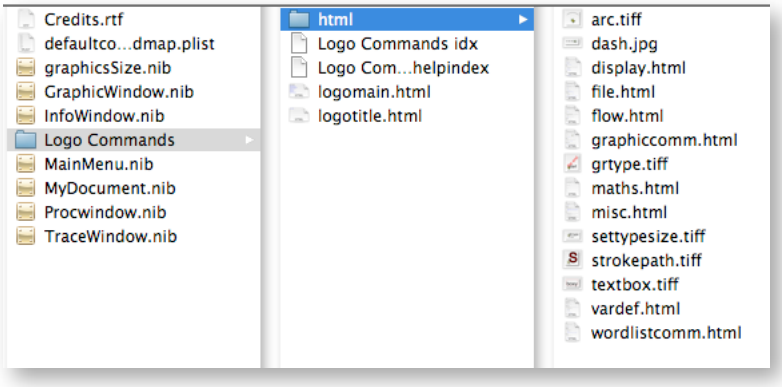


The tutorials are just files created in ACSLogo and saved in the tutorials folder. Their names have been prefixed with numbers to make them appear in a particular order. You can add, replace, rename, or change the files. The program will pick up the changes on restart.

Help

ACSLogo provides help using the standard Apple Help system. The help is contained in a set of indexed HTML files. A few pages back we looked in the application bundle and saw in the English.lproj folder a folder called *Logo Commands*.

The folder contains a folder called *html* which contains the bulk of the help HTML and image files; *Logo Commands idx* which is an index into the HTML files used by the Apple help system; a couple of other HTML files which are needed by the Help system for navigation.



If you look at one of the files in the html folder, you will see that it is valid html, with a number of added tags:

```
<!-- AppleSegStart="Back" -->
<A NAME="BACK"></A>
<A NAME="BK"></A>
<!-- AppleSegDescription="Making the turtle move backwards." -->
<h2>Back, BK</h2>
<!-- AppleSegEnd -->
<table><tr><td width=30><td><font face="sans-serif">
<P><b>Back</b> <I>distance</I></P>
<P>Move the turtle backwards <I>distance</I> pixels. Draws a line if
the pen is down.</P>
<p>Example:
<table><tr><td width=30><td><font face="sans-serif">
<p>Back 100</p>
</font></table>
<p>Opposite of <A HREF="#FORWARD">Forward</a>. See also
<A HREF="#PENUP">PenUp</a>, <A HREF="#PENDOWN">PenDown</
a><BR> </P>
</font></table>
```

There is one of these blocks for each command. The *AppleSegStart*, *AppleSegDescription*, and *AppleSegEnd* tags, and the bits they enclose, are required by the Apple Help System for indexing.

Once you've created your HTML, you need you need to run the folder through the Help Indexer found in /Developer/Applications/Utilities. You need to give it the folder, which is the *html* folder.