An Introduction to Scripting in Lingo

by Joshua Siegal

Variables

Generally speaking, in programming and scripting, variables are parts of code that stand in for certain values which may change. Usually, a variable is a word or a short collection of charaters, which can be made up by the person writing the script. Examples might be: `x` or `file1` or `var`.

Information usually gets "passed" to the variable by the use of an equals sign (`=`). **While we read in English from left to right, information in coding moves from right to left.** So, if you think of a variable as characters that represent information, in the following code, the varable `soup` is set to the value: `"navy bean"`.

```
soup = "navy bean"
```

Notice that "navy bean" is in quotes. This tells the program running the script (in this case, Director) that `"navy bean"` is a string, or assemblage of characters. The following is what you might see in Director's message box.

```
put soup

--"navy bean"
```

You have asked Director (via the `put` command) to tell you what information is contained in the variable called `soup`. It tells you `"navy bean"`, because that's what you have assigned it.

Code will execute one line after the other. See the following example:

```
soup = "navy bean"
soup = "tomato"
soup = "chowdah"

put soup
```

This will cause the message box to say:

```
--"chowdah"
```

This is because each line of code sets the variable to a different value altogether, and the last line of code that was read said that soup contained the information `"chowdah"`, and so that's what it output.

You can also set variables equal to numbers. For example:

```
x = 2
```

Then, if you wanted to add two numbers together, you could put:

```
x = 2 + 3
```

The script will compute this and assign `x` a value of `5`.

You can also do like so:

```
x = x + 3
```

If you have already set `x` to a value, lingo will take that value and add `3`; if `x` has not been set, lingo will probably assign it a value of `0` and add `3` to that. Some scripting languages will require that you specify an initial value when you create a variable, and this is a good practice.

Global Variables

Suppose we wanted to make our variable's information available to other scripts, or we wanted to make sure that the variable retained its information when the script was started over again (which occurs according to the Director frame rate). In this case, we would make our variable a global variable. Global, as in "widely available".

Unlike regular variables (which can get reset every time the script reloads), global variables must be "declared" as such. Because globals will retain their value across many different scripts, Director must know which globals will be used when it processes a given script. So, we create a list of them at the top of *each script* in which they will be used:

```
global soup
global flag1
global EZIO
```

In this case, all three terms above are global variables. This means that their information will be available to other scripts within the Director movie, and that whatever information they are assigned as the script processes will still be available when the script reloads according to the frame rate of the movie.

If you have only one handler within your script (see below for more on handlers), you can just put all your global variable declarations at the top of the script. However, if you have multiple handlers within a script, you may do better to put global declarations within the relevant handlers instead.

Integers, Floats, Booleans, and Strings

When is a 1 not a 1? When it's a string! There are many different types of information that a variable can store (or that a program can recognize). The first of these is integers, that is, whole number values. Another is floats (or floating point numbers), which are numbers with decimals. These are pretty straighforward, and Director is pretty forgiving if you mix them up, but it is good to know that in most scripting languages, they are not interchangeable, and the number 1 would not equal the number 1.00.

Another type of information that can be contained in variables is strings. It may be helpful to think of strings not as shoestrings or fishing wire, but more like a beaded necklace, since a string is a collection of (usually typographical) characters. In most scripting languages, strings must be contained within quotes. For example, if you wrote:

```
soup = chowdah
```

…then Director would not know if `chowdah` was a string or not, and since it is textual but not in quotes, Director would probably read it as another variable, one that had not been defined yet, and this might generate an error. However, if you wrote:

```
soup = "navy bean"
```

…then Director would understand that `"navy bean"` was a string, or collection of characters. Also, note that this phrase has a space in it. This is still part of the string and is considered a character like any other (maybe a clear bead in the above analogy?).

Now what if you wrote:

```
asset = "movie1"
```

…the number `1` in `movie1` is still part of a string. In this case, it represents not a quantity but the typographical charater `1`. You can use lingo functions such as `string()` to convert numbers to strings.

Another important type of variable is a boolean, named for George Boole, a 19th Century British mathematician who codified what would become the basis for computer logic. A boolean is used to show whether a property or assertion is true or false. This can also be represented by a 1 or a 0. These are very useful for evaluating code in conditional statements (more on this below). When a boolean value (`true` or `false`) is set to a variable so that a true/false condition can be tracked by the script, that variable is called a flag.

Lists

One piece of data per variable not enough for you? Try using a list! A list is a type of variable (sometimes called an array in other languages) that assigns a set of data to one variable name. That name stands in for the whole set, which in lingo is contained in brackets and separated by commas. For example:

```
ingredients = [“bread”,”peanut butter”,”jelly”,”bread”]
```

Note that bread is in there twice; that’s okay, we need two slices, so there they are. Note also that these list items are strings; in this case, the each string of text represents a sandwich-making item.

List items could also be integers, floats, other variables, etc. Here is a typical construction with integers that lists various important points in a movie file. (Director tracks movie times in 60^{ths} of a second.)

```
clippoints = [234,356,769,1156]
```

Because they’re in a list, you can then access these numbers and pass them into another statement, one after the other, or randomly, or backwards, etc.

Lingo has various functions (see below for more on functions) that allow you to manipulate your list. Some of these include:

```
count()
getAt()
random()
```

It is important to know that, behind the scenes, each position in the list has a number that Director uses to track it. While in some languages, the first position of a list or array is always 0, in lingo it is 1.

So:

```
put count(clippoints)
--4

put getAt(clippoints,4)
--1156
```

The first bit of code outputs the number of items in the list clippoints. The second outputs what information (or data) is contained in the fourth position of the list.

Here is a bit of code that finds out how many items are in the list and then picks one at random:

```
var1 = count(clippoints)
var2 = random(var1)
put getAt(clippoints, var2)
```

The first line counts the number of items in the list `clippoints` and puts that number in a variable called `var1`. The second line gets a random number between 1 (the starting point for the `random()` function) and the number in `var1`; it puts this in a variable called `var2`. The third line takes the randomly generated number in `var2` and gets the data from the list `clippoints` at that position and outputs it to the message box.

Functions

A function is a bit of code in a scripting language that performs a set of tasks, depending on the code contained in the function. The term function may also refer to the name of the function, as invoked (or "called") from another part of the code. In lingo, functions are preset and are part of the scripting language, just as verbs are part of the English language. The function `random()` in the examples above is a part of the scripting language that exists to process a certain task (generating a random number), based on what information (or "parameters" or "arguments", as that information is sometimes called) is provided to it. In some languages, a programmer may create his or her own functions, but in lingo, that is done with custom handlers (see below).

Handlers

In lingo, a handler is a type of function triggered either (a) by some other part of the code, (b) by some user interaction, or (c) by some predetermined event going on in the Director movie. That is why they are sometimes called "event handlers". They can be recognized by their use of the syntax `on`. In English, the equivalent use of this word would be something like "on resumption of talks, both sides ceased hostilities." The most familiar are probably the `on exitFrame` handler from frame scripts and the `on mouseUp` handler from sprite scripts. When triggered, the handler will execute whatever code is contained in it (between the `on` syntax and the `end` syntax).

```
on mouseUp
  clippoints = [234,356,769,1156]
  var1 = count(clippoints)
  var2 = random(var1)
  put getAt(clippoints, var2)
end
```

In the above example, which might be found in a sprite script, when the user clicks on the sprite (and the mouse button comes back up), the handler will process the code within the handler and output the value from the list.

Custom Handlers

You can also create handlers of your own and trigger (or "call") them from other areas of the script or from other scripts. For example, in a movie script:

```
on myHandler
  clippoints = [234,356,769,1156]
  var1 = count(clippoints)
  var2 = random(var1)
  put getAt(clippoints, var2)
end
```

Then, somewhere else in a spritescript, you might have a line or two that reads:

```
on mouseUp
  myHandler
end
```

This would cause the code in your handler to run only when the user clicks on the sprite in question. Why do this? Why not just include our four lines of code in the `mouseUp` handler? One reason might be that you want to trigger the same set of instructions in several different ways. Instead of repeating those lines of code each time, you put them in a handler. Then, all you have to do is "call" the handler from each trigger. Or, you might want to get a little more flexible with your handler, which is just what we'll do in the next section.

Passing Variables

Suppose you want to create a little script that does something slightly different depending on which sprite you click on. You could write a script for one sprite, copy it into all the other sprites, and then modify each one – that would work. But that might mean a lot of code. Generally, the fewer lines of code, the faster and smoother your script will run. Also, what if you make a mistake in the original (hard to believe, but we must admit the possibility). You would have to correct each of these scripts individually.

What if you could write one handler that contains all the bits of code that are common between the sprites, and send (or "pass") only the information that changes into that handler from each sprite? Of course, you can.

Let's look at an example. Suppose we have five images on the stage in our Director movie, and we want each one, when clicked on, to cue a different point in a Quicktime file that we also have on the stage. We've got each sprite in its own place in the score, so

we can reference them as sprite(1) through sprite(5), and our actual Quicktime file, let's put in sprite(10).

If we create a sprite script for the first sprite, any code in that script will apply to that sprite. In the first of the five sprites cripts, we'll put this:

```
on mouseUp
  myHandler(1)
end
```

The above example would be for sprite 1. We would do the same in the other sprite scripts, except that in the parentheses, we would use the number that corresponds to that sprite. Note: we do not have to use numbers. We could pass different words to our handler, or different numbers, etc., depending on what we wanted our handler to do. As we'll see, we will build our handler to *make use* of the numbers 1 through 5.

Now to write our catch-all handler. We would put this code not in a sprite script, but in a separate script in the Director movie, so that it can interact with all the other sprite scripts.

```
on myHandler(num)
  clippoints = [234,356,769,1156,1543]
  var1 = getAt(clippoints, num)
  sprite(10).movieTime=var1
  sprite(10).movieRate=1
end
```

The first line of this script contains the handler name itself, and a variable called num. In other scripting languages, this variable would be called an argument or parameter, but the term "passed variable" is pretty universal and least confusing. See the parentheses as the arms of the handler, grabbing the variable whose value has been passed to it by one of our sprite scripts. If it did not grab this variable, it could not make use of it within its script.

At this point, you may be thinking, "in the sprite scripts, that was an actual integer, 1 through 5, not the word 'num'". Remember that num is a variable, and that it represents any of those numbers at any given time. If we click on the first sprite, num will have a value of 1 when this script runs. If we click on the third sprite, num will have a value of 3 when this script runs, and so on. Hence the meaning of the word "variable".

The second line of this code has our list (named clippoints). This list contains movie times for sprite(10), our quicktime file. Notice that we now have five numbers in our list.

In the third line of code, we are using a varible called var1 to contain the result of the getAt function, which takes our list name, and returns the item in the list at whatever point is currently represented by num. Let's look a little closer.

We know that if we click on the third sprite, `num` will have a value of 3. So this line of the script would represent the same thing as below:

```
var1 = getAt(clippoints, 3)
```

As we know, the third item in the list is the number `769`. So, `var1` now contains the the value `769`.

However, because `num` is a variable, its value will change depending on which sprite we click on. This is the benefit of using passed variables. If we wanted to add three more sprites, we would only have to create short sprite scripts for them, similar to:

```
on mouseUp
  myHandler(6)
end
```

…and add some new numbers to the list in our custom handler. Also, if we ever want to change the behavior of the sprites when we click on them, all we have to do is change the one handler, not every single sprite script!! (Yes, two exclamation points are warranted for that.)

As for the other lines of code, assuming we clicked on the third sprite, they would set the `movieTime` property of `sprite(10)` to `769`, which is now contained in the variable `var1`. Then, they set the `movieRate` property of that sprite to 1, which is "play" or "full speed" or "100% speed". Finally, `end` ends the handler.

Conditional Statements

If only we were done with this Introduction to Scripting, then we could get on with creating some cool projects in Director!

The above is an example of a conditional statement. These statements, also known as "if-then" statements, are extremely useful in scripting. They allow the author of a script to set up certain conditions and then instruct the script to evaluate those conditions and run different blocks of code depending on the outcome.

Here are some examples in English, to get you familiar with the syntax.

```
If   my car starts this morning,   then
  I will drive to the beach.
End
```

In scripting, you must have an end to all the conditional statements, so that the script knows that no more conditions are waiting. In the above example, you have just one option if your car starts: driving to the beach.

But what if your car doesn't start! You may be a student with an extremely suspect vehicle, or you may just have bad luck. Better prepare for that possibility too.

```
If   my car starts this morning,   then
  I will drive to the beach.
Else
  I will call a mechanic.
End
```

But calling a mechanic will only fix your car. You still want to get to the beach, right?

```
If   my car starts this morning,   then
  I will drive to the beach.
Else
  I will call a mechanic.
  I will drive to the beach.
End
```

Notice that if you were actually talking, you might be tempted to say, "I will call a mechanic, then I will drive to the beach." When using conditional statements, the term then is only used in conjunction with if.

You can also "nest" conditional statements within one another.

```
If   my car starts this morning,   then
  I will drive to the beach.
Else
  I will call a mechanic.
  If   the mechanic fixes my car   then
    I will drive to the beach.
  Else
    I will sit at home and play with my dog.
  End
End
```

Each conditional statement must have its own if, then, and end.

You can also use "operators", which are "and", "or", and the like. For example:

```
If   my car starts this morning AND the weather is nice,   then
  I will drive to the beach.
Else
  I will call a mechanic.
End
```

Of course, when using the AND operator, *both* of the conditions must be true for the first statement to run. This example highlights some of the trickiness inherent in writing conditional statements. According to the above example, one logical outcome is that you wake up in the morning and start your car, but you see that it's raining, so you decide to call a mechanic. Better might be the following:

```
If   my car starts this morning OR I can borrow a friend's car,   then
  I will drive to the beach.
Else
  I will call a mechanic.
End
```

Now let's take one of the above examples and convert its syntax into something that might be useful in Director.

```
If   my car starts this morning,   then
  I will drive to the beach.
Else
  I will call a mechanic.
  If   the mechanic fixes my car   then
    I will drive to the beach.
  Else
    I will sit at home and play with my dog.
  End
End
```

Becomes

```
If  sprite(10).movieRate = 1 then
  put "movie playing"
Else
  sprite(10).movieTime = 0
  If  EZIO.readline(1) > 0 then
    sprite(10).movieRate = 1
  Else
    sprite(10).movieRate = 0
  End
End
```

So, let's break down the example above, and then we can all get on with driving to the beach and playing with our dogs. The first line evaluates whether our tenth sprite, the Quicktime movie, is playing at normal speed (has a movieRate of 1). If so, we have it sending a message to the message box (through the `put` command) that the movie is playing.

If the movie does not have a rate of 1 (it could be 0 or 0.5 or 2, etc.), then the script does a bunch of other stuff between the first `Else` and the last `End`.

That stuff: the first thing is to back the Quicktime movie up to the beginning, setting its movieTime to zero. Then begins a nested conditional statement. If the first input of the EZIO object (a physical computing interface) has a value greater than zero, the movieRate is set to 1. Otherwise, the movie is stopped (movieRate set to 0).

Here it is back in English:

```
If  sprite(10) is playing at regular speed,  then
  put “movie playing” in the message box
Else
  Back up sprite(10) to the beginning
  If  there is a signal coming from the first EZIO input, then
    Play sprite(10) at normal speed.
  Else
    Stop sprite(10)
  End
End
```

Because our second conditional statement is “nested” in the else clause of the first conditional statement, none of it will run if the first statement is true.

In other words, if the first sprite is playing at normal speed, the only thing that can come out of this script is the words “movie playing” in the message box.

Bonus question: describe the behavior of the EZIO object in this script.

Bonus question answer: If the movie is not playing at normal speed (ie, stopped or playing at some other speed) either play the sprite at normal speed or stop the sprite.

One more conditional statement:

```
If   you’ve enjoyed this little guide,   then
  I thank you for reading.
Else
  No skin off my nose, pal.
End
```