

Using Linear Lists

Working with Linear Lists

Lists are Lingo's way of implementing arrays. Like other variables in Lingo, lists are declared simply by making an assignment. For instance,

```
listOne = [ 17, 12, "Harry", 19 ]
```

declares a list called *listOne* containing the values shown.

"But wait!" cries the experienced programmer, "You can't have two data types in an array!" Well, a *list* is a little different. You may have had experience with lists in python, and although you may not have been aware of it, they allowed mixed types too. Lingo is genuinely strange in that the index of the first position is one, not zero.

Lingo is similar to other languages in that its representation of a list stores the address of the list as the variable, so assigning a list to another variable doesn't create another list, but instead just gives another "handle" for the list. Try this in the message window, and pay close attention to what happens:

```
listOne = [ 1, 2, 3 ]
listTwo = listOne
listTwo [ 2 ] = "dog"
put listOne
put listTwo
```

List Operations

One can refer to list elements by index, as shown in the previous example. There are also functions for accessing specific elements: *getAT* is a function that is passed one argument and returns the value at that position. The functions *getOne* and *getPos*, are given a value and return the first position where the value occurs. Guess what the function *getLast* does? Try using these functions in the message window.

To turn a list into a sorted list, use the *sort* function

There are also commands for adding, appending, and deleting elements. Experiment with them to get a good understanding of their behavior. The command *add* takes one argument and adds it to the end of the list (unless the list is sorted, in which case the element is inserted in order). Using the *append* command, which takes one argument, always adds the element to the end of the list. To place an element at a specific position, use the *addAt* command, which takes two arguments—first the position, then the value—and adds the value at the indicated position, shifting other elements upward to make room. Continuing the previous example, these statements will illustrate some of this:

```
listOne.add ("hambone")
put listOne
listOne.append (35)
put listOne
listOne.addAt (1, "New Beginning")
put listOne
listOne.addAt (15, "Whoa!")
```

Experiment with the various delete commands to get a sense of how they work. These include *deleteAt*, which takes one argument (the position to delete), and *deleteOne*, which takes one argument (the *value* to delete).

Some Interesting Uses for Lists

Many game-like applications require us to keep track of the movements of several objects on the screen. A simple example of this would be a screen with several objects moving downward from the top toward the bottom—think of old arcade games like “Galaga” and “Space Invaders.” One way to simplify the task is to maintain a list of the vertical positions for the sprites, and update on each frame change each one in the list. For instance, the list:

```
spriteList = [ 2, 3, 4, 5 ]
```

represents that the sprites of interest are in channels two through five. Putting the following code into the movie script would move all four sprites down the stage at an even pace:

```
global spriteList
```

```
on prepareMovie
```

```
    spriteList = [ 2, 3, 4, 5 ]
```

```
end
```

```
on prepareFrame
```

```
repeat with spriteNum in spriteList
```

```
    sprite(spriteNum).locV = sprite(spriteNum).locV + 10
```

```
end repeat
```

```
end
```

See if you can figure out how to make the sprites stop when they reach some point.

Of course, the example here is too simple even for Space Invaders. In the game, the sprites representing the individual invaders changed appearance regularly, and if the player hit one of them (more precisely, the sprite representing the player’s fire intersected the space for the invader sprite), then the invader disappeared and ceased to be able to act.

If the situation we are trying to represent is more complex, we can create a list of information we need to represent for each sprite, and then use a list of lists to handle the full set of sprites. For instance, a list that represents one sprite might include its channel number, the name of the member it currently has, and a number showing its state (0 for inactive, 1 for active). One sprite might be represented by the list

```
[ 7, “GreenMeanie”, 1 ]
```

and a list of these four item lists could represent the set of sprites. Here is one for a set of two (for space considerations):

```
[ [ 7, “GreenMeanie”, 1 ], [ 8, “BlueBomb”, 1 ] ]
```

This example still relies on the list index to identify the sprite, but we could as easily add another item to the inner lists to hold the sprite number or name.

Try expanding on the example script above to make a set of sprites move down or across the screen, alternating their appearance between two cast members, and make them stop at some point (like the edge of the screen!)

