Exercise 2: A simple scripting language

**Out:** Monday, April 16  
**Due:** Monday, April 23, 11:59 PM

# Important: please read

We would like to remind you that the following are considered cheating for this class:

* Modifying the test cases so as to cause you program to pass the tests
* Using any portion of another student’s work or allowing another student to do any portion of your work for you. You are welcome to ask one another for help. However, that help may not include the other student either typing on your keyboard or giving you code to copy and paste.

McCormick school policy requires us to report any student suspected of cheating to the Dean of Undergraduate studies for further investigation.

# Overview

In this exercise, you will add a simple scripting language to the HappyFunBlob game. We’ve already implemented the skeleton of the interpreter, including the parser, and its linkage to the game itself. So all you need to do is:

* Implement a simple dictionary class using linked lists. This will be used to hold the values of variables within the scripting language.
* Add bodies for the Run methods of the different subclasses of SyntaxTree.

As with the last assignment, we’ve provided test cases you can use to check your implementation using the automated testing tools. In addition, we’ve added event handlers to the level0.txt file, so you can try your new interpreter out in a running game.

# Getting started

This is a modified version of the code that was distributed at the beginning of the class, so be sure to use this version of the code, not the original version. To get started:

* Open up the HappyFunBlob solution file
* In the HappyFunBlob project, open the ListDictionary.cs file. This is where you’ll add your code for the first part of the assignment.

# Implementing the dictionary

Before you can implement the run methods for the interpreter, you need to implement the ListDictionary class. You’ll recall that a dictionary is a data structure that stores associations between *keys* (think of them as names) and *values*. For this assignment, you’ll implement a dictionary that

* Is limited to keys that are strings
* Allows values of any type (i.e. they’re of type object)
* Is implemented as a linked list of cells, each with one key, its associated value, and the link to the next cell.

You do not need to implement all the elaborate dictionary operations that are discussed in the book for this assignment. You need only implement:

* void Store(string *name*, object *val*)  
  Adds *name* to the dictionary with the value *val*. If name already appears in the dictionary, its entry should be modified to have the new value *val*.
* object Lookup(string *name*)  
  If the dictionary contains the key name, then it returns the value associated with it. If they key is not found in the dictionary, then it should throw a DictionaryKeyNotFoundException, by executing the following code:  
    
   throw new DictionaryKeyNotFoundException(*key*);

where *key* is the name it failed to find in the dictionary.

Once you’ve implemented your dictionary class, you can test it by selecting “Run > All tests in solution” from the Test menu” or by typing Control-R and then A. Note that this test project includes tests for the interpreter – which you haven’t even started writing yet – so don’t be surprised when those tests fail. Once you have the dictionary working, you can move on to the interpreter.

# Implementing the run methods

For this part you only need to implement one method, but you have to implement it for each of a number of subclasses of SyntaxTree. The method you need to implement is (see lecture 6):

* object Run(Dictionary dict)

This should execute the node that it is called on and return its value. If the node needs to look up the value of a variable, it can get its value from dict. The particular subclasses you need to implement are:

* **Constant**  
  Always returns the value in the Value property.
* **VariableReference**  
  Returns the current value of the variable whose name is given in the VariableName property.
* **VariableAssignment**Runs the expression in the property ValueExpression, to get its value, and sets the variable whose name is in the VariableName property to that value. Also, return this new value as the return value of Run.
* **MemberReference**  
  Runs the expression in the property ObjectExpression to get its value. Then uses the GetMemberValue() method on that object to get the member (field or property) whose name is in the MemberName property. Again, see Lecture 6 for a discussion of this.
* **MemberAssignment**  
  Run the expressions in ObjectExpression and ValueExpression, to get their values, then use the SetMemberValue() method on the result of ObjectExpression, to set it member whose name is in the MemberName property to the value returned by the ValueExpression. Also, return this new value as the return value of Run.
* **MethodCall**  
  Again, Run the ObjectExpression to get its value. You’re going to call the method of this object whose name is in MethodName. The expressions for its arguments are in the array Arguments. So iterate through all the syntax trees in Arguments, Run(dict) them, get their values, and store them in an intermediate array. Then use the CallMethod() method on the result you got from ObjectExpression to call the method, passing it the array of values you got from running all the expression in Arguments. Take the value returned by CallMethod and return that as the final value from Run().
* **OperatorExpression**  
  This is like MethodCall, but the operation we’re performing is an arithmetic operation like + or -. Like MethodCall, you will need to iterate over the argument expressions, calling Run on them to get their values. In this case, all the child nodes are arguments, so you can just use the property Children to get an array of all the arguments. As discussed in lecture 6, we’re provided you with a procedure, Interpreter.GenericOperator, that will do the actual arithmetic for you, regardless of what types the arguments are. Having computed the arguments, just return the value of:  
    
   Interpreter.GenericOperator(Label, *arguments*)  
    
  where *arguments* is an array containing the results you got from calling Run on all the Children. The property Label contains the name of the operation to perform (e.g. “+”, or “-“), so pass it along as the first argument.

All these classes can be found in the file SyntaxTree.cs. We’ve implemented everything in the class for you except the bodies of the run methods. Once again, you can run the test suite in the Test project to validate your implementation.

# Trying it out

We’ve hacked the game code so that when you can specify handlers for different events in the level files. If you right click on HappyFunBlobGame and select “Set as startup project”, then the run button will run the game rather than the tests. You should fine that when you run the game, the following events are handled by scripts:

* Touching the first orb displays the message “Touched orb 1”
* Letting go of the first orb displays the message “Untouched orb 1”
* Same for the second orb, only it says “orb 2” rather than “orb 1”
* Successfully landing on the final orb turns it from red to green

You can see the scripts that do this by looking in the level0.txt file (best done with Excel, but notepad will work too – see Lecture 6 for instructions). In the level files, the events get treated as fields that you “store” scripts into. So for example, the line:

Start Orb 50 40 50 Touched game.Display(100, 100, 0, "Touched Orb 1")

To put a particular script in the “touched” event for the orb. In particular, it tells the system that when the orb is touched, it should run the script code:

game.Display(100, 100, 0, “Touched Orb 1”)

In other words, it should call a particular method of the object in the variable game. In this case, the game variable is the HappyFunBlobGame object. We’ve added a method called Display to the object to let you print things on the screen from within a script:

* game.Display(*x*, *y*, *time*, *string*)  
  Displays the *string* at coordinates (*x*, *y*) for *time* seconds. If time is 0, the message is displayed until it is replaced by the next call to Display.

# Turning it in

As before, to turn the assignment in, you should:

* Choose “Clean solution” from the Build menu in Visual Studio. This will remove binary files and bring your directory down to a manageable size.
* Exit Visual Studio
* Make a ZIP file of your assignment’s directory. Please use ZIP format, not RAR, tgz, or other formats.
* Upload the zip file to blackboard. Remember that blackboard may not work with Firefox or Google Chrome. So if you get a message saying you need to choose a “valid file”, try submitting it using Internet Explorer.

# 