Exercise 2: A simple interpreter

**Out:** Satuday, April 25  
**Due:** Friday, May 1, 12 noon

# 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 interpreter, such as could be used for a scripting language for an application. We’ve already implemented the skeleton of the interpreter, including the parser, and a simple GUI (graphical user interface) for testing it. 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.

# Getting started

To start the assignment:

* Open up the Interpreter solution file (Interpreter.sln)
* In the Interpreter 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.

* int Count { get; }

The is a property with just a get method. It should return the number of items stored in the dictionary. Note: You must write this in such a way that it takes time. That is, you should **not** just write a loop that counts the number of items. Instead, store the number of items as a separate field, and update it each time an item is added.

* The for loop for enumerating elements of the list.  
  This is in the found at the end of the file in the method called GetEnumerator. All you need to do for this is to remove the NotImplementedException, uncomment the for loop, and change the for loop to use the particular variable and field names that you used for your ListDictionaryClass.

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. See Lecture 6 and the discussion of reflection, below, for an explanation of this.
* **MemberAssignment**  
  Run the expressions in ObjectExpression and ValueExpression, to get their values, then use the SetMemberValue() method (again, see the discussion of reflection) 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.

# Reflection

This assignment introduces a new programming technique you probably don’t have experience with, called “reflection.” Reflection lets you take an object and

* Ask what its type is
* What the names of its fields are
* To get and set the values of those fields
* To ask what its methods are
* And to call those methods

The reflection interfaces for Java and C# are more complicated than is really necessary, so we’ve provided you with a simplified interface:

* *object*.GetMemberValue(*string* *fieldName*)  
  Takes *object*, looks inside it to find the field named *fieldName*, and returns its value.
* *object*.SetMemberValue(*string* *fieldName*, *object newValue*)  
  Same, but changes the field’s value to the specified newValue.
* *object*.CallMethod(*string* *methodName*, *object[] methodArguments*)  
  Calls the method named *methodName* on *object*, passing it the arguments in *methodArguments*. If *methodArguments* has 1 element, the method is called with 1 argument, if it has 2 elements, it’s called with 2 arguments, etc. Note: if you took 111, this is in some ways like C#’s version of apply.

# Trying it out

You can run the interpreter by choosing “Start Debugging” (F5) from the Debug menu. This will pop up a window that lets you type expressions and see their results and the values of your variables. And once again, you can run the test suite in the Test project to validate your implementation.

# Turning it in

To turn the assignment in, you should:

* Make sure the tests run properly.
* Make a new folder called Assignment 2 (include the space in the name).
* Copy just the ListDictionary.cs and SyntaxTree.cs files into it.
* Compress the Assignment 2 folder into a zip file. You should end up with a file called “Assignment 2.zip”.
* Upload the file to Canvas.
* Congratulations, you’re done!

# 