Exercise 2: A simple interpreter

Out: Satuday, April 15
Due: Friday, April 22, 12 noon

Important: writing unit tests

For this assignment, we will not be providing you with unit tests. You might be tempted not to write your own on the grounds that the code "seems to work." We can't force you to write tests, but take it from us, it's a serious risk not to test your code properly. You will not have the opportunity to resubmit a correction if your code does not pass our tests.

Overview

In this exercise, you will write 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 Expression.

Unlike the previous assignment, we will not be providing you with unit tests for your code. We strongly recommend that you write your own, however. We will have our own unit tests that we will use for grading.

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 $O(1)$ 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 should write some unit tests for it. Go to the Tests project and open the ListDictionaryTest.cs file. And fill in some TestMethods for it. The last assignment talked about how to write unit tests in C#, but we've included a more detailed discussion in the appendix to this assignment.

To test your ListDictionary class, you'll need (at a minimum) to write test methods that make a ListDictionary, put some data in it using Store(), and try to read the data out again using Lookup(). You can then use the Assert.AreEqual() method to check to make sure that Lookup() returned the correct value.

Once you've written your tests, you can run them by selecting "Run > All tests in solution" from the Test menu" or by typing Control-R and then A. 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 Expression. 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 Expressions 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 Subexpressions 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 Subexpressions. 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 Expression.cs. We've implemented everything in the class for you except the bodies of the run methods.

Each time you implement a run method, you should add at least one test method for it to the appropriate file in the Tests project. Then run your tests. Again, the way to test your classes is to make an instance of the class you're testing placing the appropriate data inside of it, and then calling its Run() method, and use Assert.AreEqual() to check whether the run method returns the correct value.

You can make an instance of your expression subclass either by calling its constructor directly (e.g. typing "new Constant(7)". Or by calling the Expression.Parse() method, which takes a string containing source code as an argument. For example:

Expression.Parse("a+1");

will return a parse tree with an OperatorExpression for the addition, along with a VariableReference and a Constant as its Subexpressions.

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

Start by running the unit tests you've writing using the Test>Debug>All tests or Test>Run>All tests menu commands, as before. Once your code passes its unit tests, you can do integration testing by running the interpreter with its GUI. 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.

Turning it in

To turn in your assignment, please do the following:

- In Visual Studio, chose "Clean Solution" from the Build menu. This will remove all the binary files from the solution, leaving only the source code. That will make your assignment smaller and easier to upload.
- Close Visual Studio
- Right click on the folder icon for your assignment and choose the "Compressed (zipped) folder" selection from the Send To menu.
  - **IMPORTANT:** Many of you are using commercial compression software such as **WinRar**. You may use these programs to make a zip file, if you prefer. However, **you must specifically turn you program in as a ZIP file**. Do not turn in other file formats such as RAR, TAR, TGZ, etc.. **Non-ZIP files will be returned ungraded** and your assignment will be **counted as late until you submit a ZIP file**.
- Return to this assignment on Canvas and upload your zip file as your submission.

Appendix: writing tests

Each test is just a simple method that calls your code and makes **assertions** about what should happen as a result. As in EECS-211, an assertion is a piece of code that checks a condition. If the condition is true, then execution continues, but if it's false, the code throws an exception. The testing system works by running each test method, and keeping track of which of them has assertions that fail. For those whose assertions fail, it remembers the assertion that failed and displays it alongside the name of the assertion in the list at the bottom of the screen.

Microsoft's testing framework supports a number of assertions, all of which are methods of the magic class Assert:

- Assert.IsTrue(boolean)
  Continues execution if it's argument is true, otherwise fails and marks the test as having failed.
- Assert.IsFalse(boolean)
  Same, but the test passes if the boolean is false.
- Assert.AreEqual<Type>(x, y)
  Test passes if x and y are equial. The <Type> part tells what type x and y are.
- Assert.Fail()
  Never passes. It's a way of saying "execution should never get to this point, so if it does, then something's wrong".

These methods can optionally take an error message as an additional argument, in which case the testing rig will display that message rather than a generic "test failed" message, should the assertion fail.

Marking test methods using attributes

When the test system starts up, it will look at all the files in the Test project and search for methods that start with the annotation:

```
[TestMethod()]
```

Or:

```
[TestMethod]
```

It finds all those methods and runs them. The bracketed annotations are called **attributes** and they they allow the code to search through its own methods at run time to find ones that are annotated with particular attributes. So the equivalent of (check-expect *expectedValue expression*) from EECS-111 is something like this:

```
[TestMethod]
public void nameofmytest() {
   Assert.AreEqual(expectedValue, expression);
}
```

So it's very much like 111, only more verbose. Sometimes you need to have some setup code before you do the real test, for example to make some local variables or do some initialization of an object. If so, you can put that in the method, before the call to Assert.AreEqual:

```
[TestMethod]
public void nameofmytest() {
   … setup code …
   Assert.AreEqual(expectedValue, expression);
}
```

For example, here are some of the tests for array queues from assignment 1:

```
[TestMethod()]
public void ConstructorCreatesEmptyQueueTest()
{
    ArrayQueue target = new ArrayQueue();
    Assert.IsTrue(target.IsEmpty,
            "A newly created ArrayQueue should have IsEmpty=true, but doesn't");
}

[TestMethod()]
public void ConstructorCreatesNonFullQueueTest()
{
    ArrayQueue target = new ArrayQueue();
    Assert.IsFalse(target.IsFull,
            "A newly created ArrayQueue should have IsFull=false, but doesn't");
}

[TestMethod()]
public void QueueEventuallyFillsText()
{
    ArrayQueue target = new ArrayQueue();
    int i;
    for (i = 0; i < 1000000 && !target.IsFull; i++)
        target.Enqueue(i);
    Assert.IsTrue(target.IsFull,
    "Added 1000000 elements to ArrayQueue and it never registered IsFull=true");
}
```

Notice that these tests specify custom error messages to print, since these are more informative than a generic “test failed” message. But you don’t have to do that for your code if you don’t want to.

Sometimes, you want to write a test to make sure that a method that’s supposed to those an exception does throw an exception. You can do that by adding an ExpectedException attribute:

```
[TestMethod]
[ExpectedException(typeof(ExceptionType))]
public void nameofmytest() {
    … setup code …
    … code that ought to generate the exception…
}
```

The ExpectedException attribute says that the test passes if it throws the specified exception. The Assert.Fail() call says that if the code gets that far, the test has failed (because it didn’t throw the exception). For example, here is one of the tests for the array queues in assignment 1:

```
[TestMethod()]
[ExpectedException(typeof(QueueFullException),
                   "Adding to full queue should throw QueueFullException")]
public void EnqueueToFullQueueThrowsQueueFullExceptionTest()
{
    ArrayQueue target = new ArrayQueue();
    Assert.IsFalse(target.IsFull);
```

```
            int i;
            for (i = 0; i < 1000000 && !target.IsFull; i++)
                target.Enqueue(i);
        }
```

Again, this test is a little fancier in that it includes extra arguments to ExpectedException to provide custom error message for this particular test. Also, the real test we handed out in the code included some more checks to provide still better error messages, but this gives you the general idea.