Expression Tree Code Demo (cont.)

Cpt S 321

Washington State University

Let’s see some solutions

What we did last time:

3. Consider operator precedence/associativity

4. Parse the expression string and build the expression tree more elegantly

Lambda expressions

- Lambda expression: unnamed method written in place of a delegate instance
- Form: (*parameters*) => *expression*
  (*parameters*) => *{ sequence of statements };*
  - no parentheses needed when one parameter only
- Example:

```
1. delegate int Transformer (int i);      // declaring a delegate
2. Transformer sqr = x => x * x;          // instantiating the delegate with a lambda expression
3. Console.WriteLine(sqr(3));             // will output 9
```

Lambda expressions (cont.)

- You can specify the types of lambda parameters explicitly:
  int x => x * x;

- Lambda expressions that don't return anything can be assigned to the type Action<T> (delegate within the System namespace)
  - Action<int, int> action1 = (x, y) => Console.WriteLine((x + y).ToString());
  - Action action2 = () => Console.WriteLine("Hello!");

- Lambda expressions which return values can be assigned to Func<T, TResult> (delegate within the System namespace)
  - Func<int, int, bool> testForEquality = (x, y) => x == y;
  - Func<int, string, bool> isTooLong = (int x, string s) => s.Length > x;

Lambda expression as input to a function

```
public static List<T> TransformList<T>(List<T> inputList, Func<T,T> transform)
{
    List<T> transformedList = new List<T>();
    foreach (T item in inputList)
    {
        transformedList.Add(transform(item));
    }
    return transformedList;
}
```

LINQ (or Language Integrated Query)

- Allows us to write queries over local object collections and remote data sources
- We can query any collection that implements **IEnumerable<T>**
- Basic unit of data in LINQ:
  - Sequences: A sequence is any object that implements IEnumerable<T>
  - Elements: An element is each item in the sequence

LINQ (cont.)

- Query operator: A method that transforms a sequence. It operates on an input sequence and results in an output sequence (the input sequence is not altered)
- Examples of the standard query operators defined in System.Linq: Where, Count, OrderBy, ThenBy

- More on MSDN:
  - Standard Query Operators Overview
  - Enumerable Class

LINQ (cont.)

- Simple query example:

```
string[] names = { "Tom", " Tim", "Harry" };
IEnumerable<string> filteredNames = names.Where(n => n.Length >= 4);
// Equivalent to:
// IEnumerable<string> filteredNames = System.Linq.Enumerable
                                    .Where (names, n => n.Length >= 4);
foreach (string n in filteredNames)
{
    Console.WriteLine (n);
}
```

- Output:

```
Harry
```

LINQ (cont.)

- Example when chaining queries:

```
string[] names = { "Tom", "Tim", "Harry", "Mary", "Jay" };
IEnumerable<string> query = names
    .Where (n => n.Contains ("a"))
    .OrderBy (n => n.Length)
    .Select (n => n.ToUpper());
foreach (string name in query) Console.WriteLine (name);
```

- Output:

```
JAY
MARY
HARRY
```

Solutions we identified last time

- Throw more descriptive exceptions
- ~~2. Get rid of the hardcoded operators~~
- Allow support for new operators without needing to change the logic in every method
- ~~1. Extract classes into separate files~~
- ~~3. Consider operator precedence/associativity~~
- ~~4. Parse the expression string and build the expression tree more elegantly~~
- Get rid of the redundant code

Where next?

Pointers for solutions ExpressionTreeCodeDemo

- **Allow support for new operators that are added after we have shipped our code, i.e., our code does not need to be changed.**

- We will allow the user to define their own operators. User defined operators must:
  - Inherit from our **OperatorNode** class
  - Implement three properties (**Operator, Precedence, Associativity**) and one method (**Evaluate()**)

- To handle this, we will use Reflection and LINQ: We will load all classes that implement our **OperatorNode** class and add it to our dictionary of handled operators:

```
private Dictionary<char, Type> operators = new Dictionary<char, Type>();
```

Reflection

- In our OperatorNodeFactory class we will implement a delegate OnOperator:

  private delegate void OnOperator(char op, Type type);

- Define method TraverseAvailableOperators with the following signature and leave it empty for now, we will implement it in a couple of steps:

  private void TraverseAvailableOperators(OnOperator onOperator)

- In the constructor of the OperatorNodeFactory, we will instantiate the delegate with a method that adds the operator character-type pair to the dictionary and we will call method TraverseAvailableOperators with the instance of the delegate. Using **lambda expressions**, we can do this in one step:

  TraverseAvailableOperators((op, type) => operators.Add(op, type));

```
using System.Reflection;
...
```

Implement TraverseAvailableOperators

```
private void TraverseAvailableOperators(OnOperator onOperator)
        {
            // get the type declaration of OperatorNode
            Type operatorNodeType = typeof(OperatorNode);

            // Iterate over all loaded assemblies:
            foreach (var assembly in AppDomain.CurrentDomain.GetAssemblies())
           {
                // Get all types that inherit from our OperatorNode class using LINQ
                IEnumerable<Type> operatorTypes =
                     assembly.GetTypes().Where(type => type.IsSubclassOf(operatorNodeType));
```

LINQ

Implement TraverseAvailableOperators (cont.)

```
  // Iterate over those subclasses of OperatorNode
 foreach (var type in operatorTypes)
{
    // for each subclass, retrieve the Operator property
    PropertyInfo operatorField = type.GetProperty("Operator");
    if (operatorField != null)
    {
          // Get the character of the Operator
          object value = operatorField.GetValue(type);
          // If your “Operator” property is not static, use the following code instead:
          // object value = operatorField.GetValue(Activator.CreateInstance(
               type, new ConstantNode("0"), new ConstantNode("0")));
          if (value is char)
          {
              char operatorSymbol = (char)value;
              // And invoke the function passed as parameter
              // with the operator symbol and the operator class
              onOperator(operatorSymbol, type);
           }}}}}
```

Change the implementation of the factory method

```
public OperatorNode CreateOperatorNode(char op)
 {
     if (operators.ContainsKey(op))
     {
         object operatorNodeObject = System.Activator.CreateInstance(operators[op]);
         if (operatorNodeObject is OperatorNode)
         {
             return (OperatorNode)operatorNodeObject;
         }
     }
     throw new Exception("Unhandled operator");
 }
```

What did we do?

- We replaced the hard coded operator with a function that will populate the dictionary of operators based on classes that derive from OperatorNode.
- How to test?
  1. Remove any trace of hardcoded operator in the initialization of the dictionary.
     Run your tests – they should all pass!

  2. Let's test how easy it is to extend the available operators **from another project**
     - Ex: In a DLL project (OperatorLibrary), create a class **MyPowerOperator** (for mine I use '^', right associative, precedence 2)

  3. **Add more tests with your new operator and make sure they pass**

     In your **test project:**
     - Add the .dll file of the library project (OperatorLibrary.dll) to the test project (in bin/Debug)
     - make sure you load the new DLL project in your tests so that the traverse method can look in it: **Assembly.Load("OperatorLibrary");**