Implementing nondeterminism in a deterministic language

Overview

In this exercise, you will use the continuation passing technique discussed in the lecture to implement nondeterministic search for a simple recursive-decent parser for context-free grammars. We've implemented the framework classes and a unit testing rig for you. You need only fill in the `Parse()` methods of the code.

Don't try to read this assignment until you've watched lecture on non-determinism.

Software used for this assignment

For this assignment, you will use Visual Studio and .NET 5. If you are on a PC, use Microsoft Visual Studio Community. I'm using Visual Studio Community 2019, but newer versions will work too. If you're on a Mac, use Visual Studio Mac. You may also need to download and install .NET 5; I needed to install it for my Mac, but for my PC, it seemed to be there already.

Starting up

Open the file Parser.sln. That should launch Visual Studio and open the code. It will have two projects in it, "Parser", which is the real code, and "ParserTests", which is the unit tests we've written for you.

Running unit tests

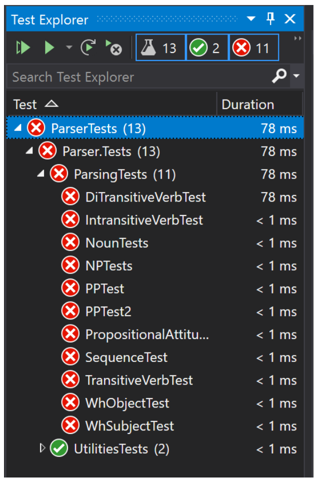
To run the unit tests, open the test window (called "test explorer"). In windows, go to the **Test** menu and choose **Test explorer**. On Mac, go to the **View** menu and choose **Tests**.

Running all tests

Press the button to run all tests. On windows, this looks like:

On mac, it looks like:

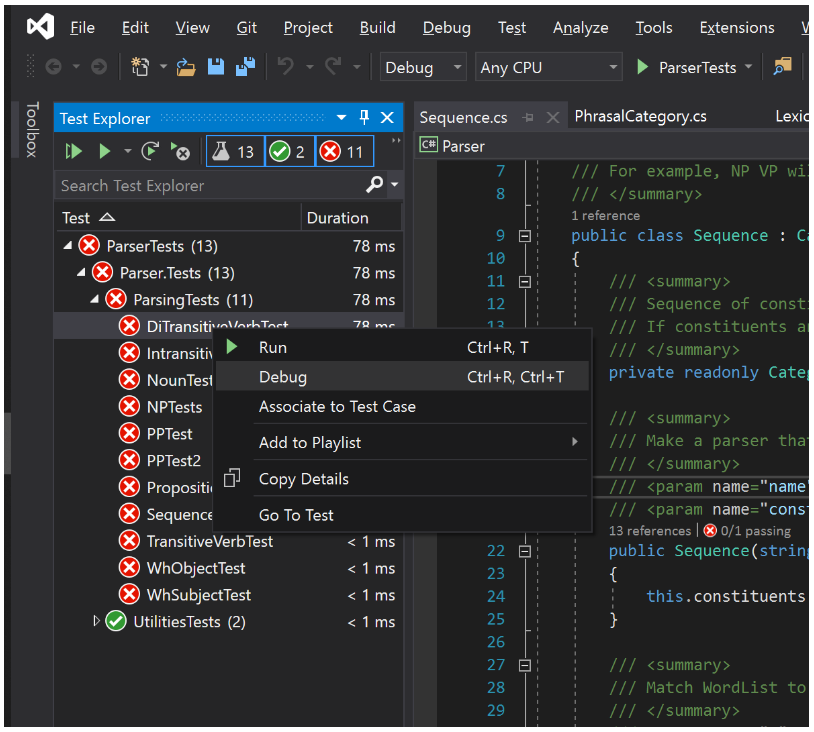
You'll see green check marks next to the tests that worked and red Xs next to those that failed. The Parse() methods for the classes are all written to throw exceptions at the moment, so all the parsing tests will fail and only the tests of the utility procedures will work:



```
Test                          Duration
◢ ParserTests (13)            78 ms
  ◢ Parser.Tests (13)         78 ms
    ◢ ParsingTests (11)       78 ms
        DiTransitiveVerbTest  78 ms
        IntransitiveVerbTest  < 1 ms
        NounTests             < 1 ms
        NPTests               < 1 ms
        PPTest                < 1 ms
        PPTest2               < 1 ms
        PropositionalAttitu…  < 1 ms
        SequenceTest          < 1 ms
        TransitiveVerbTest    < 1 ms
        WhObjectTest          < 1 ms
        WhSubjectTest         < 1 ms
    ▷ UtilitiesTests (2)      < 1 ms
```

You can ignore the utility tests; they're just the tests I used when I was writing them.

Select one of the tests that's failing in the test window. Right click and select Debug test:



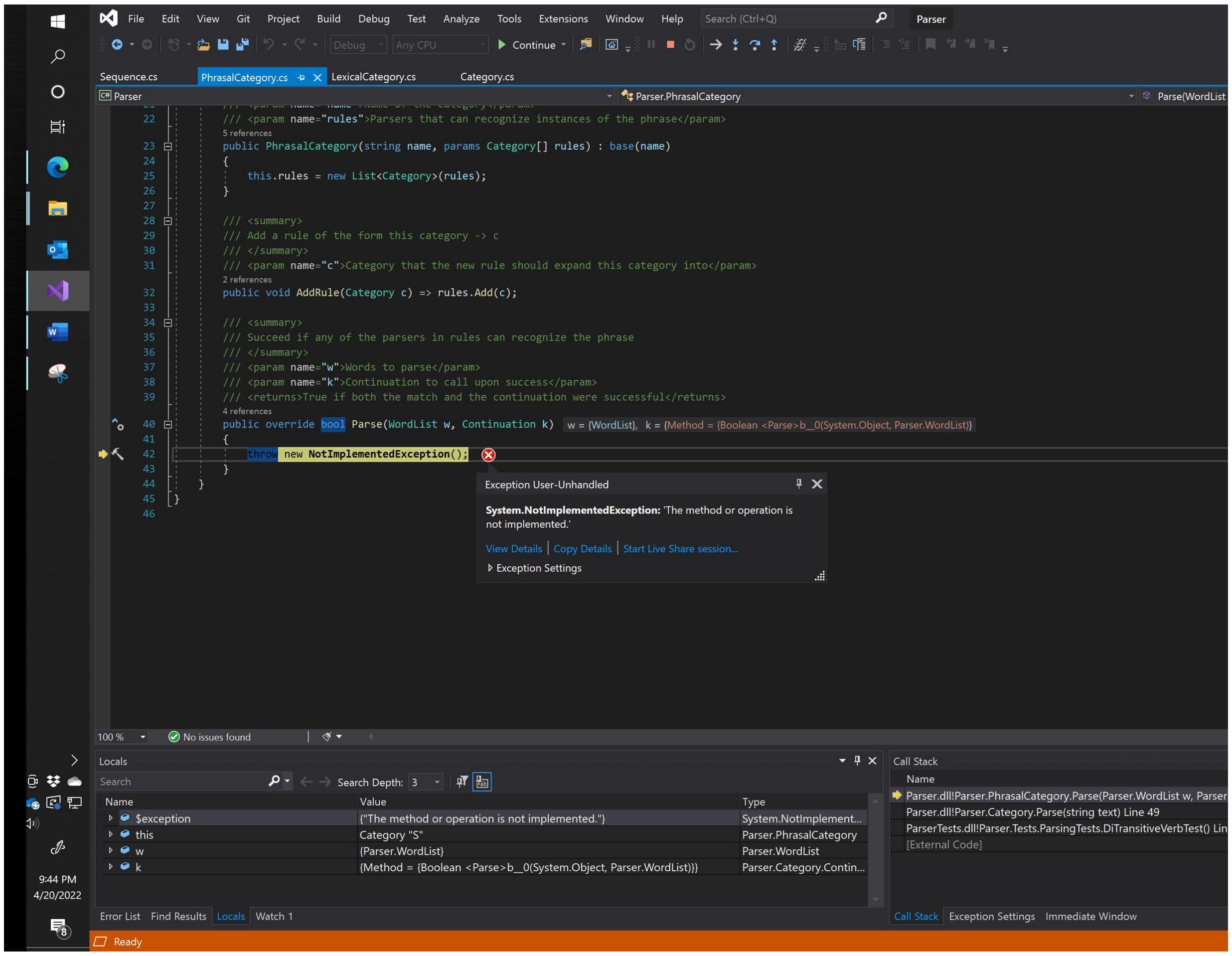
```
Test                          Duration
◢ ParserTests (13)            78 ms
  ◢ Parser.Tests (13)         78 ms
    ◢ ParsingTests (11)       78 ms
        DiTransitiveVerbTest  78 ms
        Intransiti
        NounTest
        NPTests
        PPTest
        PPTest2
        Propositi
        Sequence
        TransitiveVerbTest    < 1 ms
        WhObjectTest          < 1 ms
        WhSubjectTest         < 1 ms
    ▷ UtilitiesTests (2)      < 1 ms
```

```
/// For example, NP VP wi
/// </summary>
public class Sequence : C
{
    /// <summary>
    /// Sequence of const
    /// If constituents a
    /// </summary>
    private readonly Categ

    /// <summary>
    /// Make a parser tha
    /// </summary>
    /// <param name="name
    /// <param name="cons
    public Sequence(strin
    {
        this.constituents
    }

    /// <summary>
    /// Match WordList to
    /// </summary>
```

It will run just that test with debugging enabled, so you'll be able to set breakpoints, look at the call stack, and so on. Since we haven't set any breakpoints, this will just run until the point where it throws the exception:



```
        /// <param name="rules">Parsers that can recognize instances of the phrase</param>
        public PhrasalCategory(string name, params Category[] rules) : base(name)
        {
            this.rules = new List<Category>(rules);
        }

        /// <summary>
        /// Add a rule of the form this category -> c
        /// </summary>
        /// <param name="c">Category that the new rule should expand this category into</param>
        public void AddRule(Category c) => rules.Add(c);

        /// <summary>
        /// Succeed if any of the parsers in rules can recognize the phrase
        /// </summary>
        /// <param name="w">Words to parse</param>
        /// <param name="k">Continuation to call upon success</param>
        /// <returns>True if both the match and the continuation were successful</returns>
        public override bool Parse(WordList w, Continuation k)
        {
            throw new NotImplementedException();
        }
    }
}
```

```
Exception User-Unhandled

System.NotImplementedException: 'The method or operation is not implemented.'
```

```
Locals
Name          Value                                                           Type
$exception    {"The method or operation is not implemented."}                 System.NotImplement...
this          Category "S"                                                    Parser.PhrasalCategory
w             {Parser.WordList}                                               Parser.WordList
k             {Method = {Boolean <Parse>b__0(System.Object, Parser.WordList)}} Parser.Category.Contin...
```

```
Call Stack
Name
Parser.dll!Parser.PhrasalCategory.Parse(Parser.WordList w, Parser
Parser.dll!Parser.Category.Parse(string text) Line 49
ParserTests.dll!Parser.Tests.ParsingTests.DiTransitiveVerbTest() Lin
[External Code]
```

Pressing the stop button in the toolbar or selecting **Stop Debugging** from the **Debug** menu will stop the debugger and let you go back to editing.

That's all there is to do with tests until you start writing code.

Class hierarchy

Speaking of which, let's talk about the structure of the code.

The parser is implemented as a class hierarchy, with classes implementing kinds of syntactic categories:

- **`Category`**
  This is the base class for all the categories. All Categories have:
  - Name (a string)
    This is used partly for debugging purposes, but it also gets included in the parse trees so you can tell what category a complex phrase is.
  - `Parse()` method
    We'll say more about these below. This is the main thing you'll be writing.

- **`LexicalCategory`**
  This is the class for lexical categories, i.e. categories of words. It has a hashset[1] of words inside it (called, uncreatively, `words`). Its `Parse()` method should test whether the next word in the input is in the list of words. It succeeds (calls its success continuation) if it is, and fails (returns false) if not.
- **`Sequence`**
  This is a kind of pseudocategory that represents all the phrases made up as a sequence of other categories. It contains an array of those categories, in order, called `constituents`. Its `Parse()` method should succeed if the input starts with instances of those constituents, in order, and otherwise it should fail.
- **`PhrasalCategory`**
  This is the class for phrasal categories, i.e. categories of phrases. It has a list of "rules" for phrases it can be expanded into (again, the list is called `rules`). However, since all rules are of the form $category \rightarrow phrase$, and all the rules for a given category start with the same $category \rightarrow$, the `rules` is really just list of the phrase parts of the rules. And in fact, the phrases are really just represented as categories, since we have the `Sequence` class as a kind of fake category type for representing sequences.

For example, if we wanted to implement the rules defined by:

$$NP \rightarrow Pronoun$$
$$NP \rightarrow Det\ N$$

We would first create the lexical categories Pronoun, Det, and N:

```
public static readonly Category Pronoun = new LexicalCategory("Pronoun", "I", "you", …);
public static readonly Category Det = new LexicalCategory("Det", "the", "a", …);
public static readonly Category N = new LexicalCategory("CommonNoun", "cat", "dog", …);
```

Note that the first argument of the constructor is always a string that gives the name of the category; without that, it's impossible to tell what's going on in the debugger.

Given the lexical categories, you can then define the phrasal category NP as:

```
public static readonly Category NP = new PhrasalCategory("NP",
                                                         // NP → Det N
                                                         new Sequence("NP", Det, N),
                                                         // NP → Pronoun
                                                         Pronoun);
```

We've included a grammar for a tiny subset of English in the file Grammar.cs, so you don't need to define a grammar yourself.

[1] A hashset is a hashtable that just has keys; not values associated with them. So the operations are just adding/removing elements, and checking if a value is an element of the table.

Grammar

The grammar the tests use is given in the file **Grammar.cs**. It's written as a static class, so it basically just contains a bunch of static fields to hold the different Category objects (NP, VP, etc.).

Take a look at the file. This is a fancier grammar than we talked about in lecture. Although it doesn't handle adjectives, it handles questions, prepositional phrases, and verbs that take "sentential complements", meaning their "objects" are sentences or at least verb phrases. "Believe" is such a verb: in the sentence "I believe I understand sentential complements", the sentence "I understand sentential complements" is the object of the verb "believe".

One complication, id that this means that in this grammar, a VP can contain an S, but an S can of course can contain a VP. The OptPP category is also recursive that way.

This means the Category objects form a cyclic graph and making a cyclic graph is very slightly complicated. You can safely skip this paragraph since you don't need to alter the grammar for the assignment. I'm explaining it here just because if I don't some conscientious student will worry about it and get confused. So here's how it works: since the declarations of the static fields can only use the values of fields declared before them, we have to create VP and OptPP first without the cyclic references, and then later, at the end of the file, is a static constructor for the class that adds in the extra, recursive, rules once all the objects have been created. That's the reason there's an AddRule method for PhrasalCategory but no add method for anything else. It's also why PhrasalCategory stores its rules in a list rather than an array like the other classes.

Writing the `Parse()` methods

We've written all the boilerplate code for you so you can focus on writing the parser itself. The good news is that it's only a few lines of code, but the bad news is that it involves the use of explicit continuations that we discussed in lecture.

The parser is implemented as a set of `Parse()` methods in the different subclasses of `Category`. These methods are called with a list of words and check to see if the list of words starts with an instance of the category. If so, it calls its success continuation with a "parse" of the subphrase corresponding to that category and the remaining words (the ones after the subphrase). So for example,

- If we call Pronoun.Parse on "I ate the cake", it will call its success continuation with "I" as the parse and "ate the cake" as the remaining words.
- If we call Pronoun.Parse on "the cat ate the cake", it will fail because that sentence doesn't start with a pronoun.
- If we call NP.Parse "I ate the cake", it will call its success continuation with ["NP" "I"] as its parse (more on the representation of parses later), and "ate the cake" as the remaining words.
- If we call NP.Parse "the cat ate the cake", it will call its success continuation with ["NP" "the" "cat"] as its parse (more on the representation of parses later), and "ate the cake" as the remaining words.
- If we capp VP.Parse on "ate the cake", it will call its success continuation with ["VP" "ate" ["NP" "the" "cake"]] as its parse, and null (the empty list) as the remaining words.

The type signature of `Parse()` is:

```
    public abstract bool Parse(WordList w, Continuation k);
```

Where `WordList` is a linked list of words:

```
    public class WordList
    {
        /// <summary>
        /// The first word in the list
        /// </summary>
        public readonly string FirstWord;

        /// <summary>
        /// Remaining words in the list
        /// </summary>
        public readonly WordList Rest;
    }
```

Continuations

Our `Continuations` will be procedures that take an object and `WordList` as input and return a Boolean:

```
    public delegate bool Continuation(object parse, WordList remaining);
```

To understand what these arguments are, remember that the `Continuation` is the thing a `Parse()` method calls when it's successful. So its inputs (arguments) are effectively the outputs (return values) of the parser. Here, the parser is "returning" two values:

- The remaining words that it didn't use in its parse, if any (the `WordList` argument)
- The parse of the phrase, that is, its division into subphrases and the categories of those subphrases. This is what we'll use to tell if the parser is working. And in a real natural language understanding system, it would be the data structure that a later system used to understand the meaning of the sentence.

Why does the Continuation return a Boolean? Because we need some way to allow it to reject the parse and ask for a different one. So it returns true if it likes the parse and false if it wants to backtrack.

Representing parse trees

We'll represent parse trees as:

- **`LexicalCategory`**
  The parse is just whatever word the parser identified, i.e. just a string.

- **`Sequence`**
  An object[] array containing
  - The Name field of the sequence. This will let us tell whether something got parsed as an NP, a VP, an S, or whatever.
  - The other elements are whatever the parse outputs are for the constituents of the phrase

  So the parse output for the sequence new Sequence("Foo", Det, N) for the words "the cat" would be the array with elements:

    - "Foo"
    - "the"
    - "cat"

  The testing code will print this as the string: "[Foo the cat]" as an approximation to the representation used in lecture.

- **`PhrasalCategory`**
  The parse output from a PhrasalCategory should just be the parse output from whatever rule worked. So for our example category, `new PhrasalCategory("NP", new Sequence("NP", Det, N),Pronoun)`, the parse output for "I ate the cake" would be just the string "I", since that's what Pronoun outputs, but the parse output for "the cat ate the cake" would be an array with the elements "NP", "the" and "cat", because that's what the output of the Sequence would be.

What to write

So now fill in the Parse methods for the LexicalCategory, Sequence, and PhrasalCategory methods. This only requires you to write a few lines of code, but they are somewhat hairy lines of code if you haven't done this before.

Important: this code will be autograded. When we test your code, we will only use your copies of **LexicalCategory.cs**, **Sequence.cs**, and **PhrasalCategory.cs**. So we don't recommend changing the other files. It's fine to add new tests if you want, or even to modify the other files. Just understand that we will be testing with unmodified copies of those files. You need to make sure your code compiles and runs with unmodified copies of the other files, or you will likely get a zero for the assignment.

LexicalCategory

Start with the `Parse()` method for the `LexicalCategory` class. This only needs to look at the first word of the input and check if it is in the `words` set. Use can use the `Contains()` method of the `words` hashset to test whether a given word is in the category. If it is, call the success continuation and the remaining words and return its value. Otherwise, return false.

Run the unit tests again. The NounTests test should now pass.

Sequence

Now write the `Parse()` method for `Sequence`. This is the complicated one. It needs to call the parse method for the first constituent, passing it as a continuation something that will call the parse method of the second constituent with a continuation that calls the parse method of the third constituent, and so on. So you need to write this as a recursion and you'll need a helper function. IT IS NOT CORRECT TO WRITE THIS AS A LOOP. Doing so will break backtracking, since you need to have calls to all the constituents on the stack in case you need to backtrack.

At the end, the parse you pass on to the continuation (assuming the parse is successful) should be an array starting with the Name field of the sequence and continuing with the outputs of the parser for each constituent. In my solution set, I did this by have the Parse method for Sequence make the array, then fill it in as each constituent is parsed.

Run the unit tests again. The SequenceTests test should now pass.

PhrasalCategory

Now fill in the `Parse()` method for `PhrasalCategory`. This one is in some ways the easiest. It should just try the parsers for all the `rules` until one works, and pass whatever outputs it generates (parse and remaining words) to its own continuation.

Run the tests again. When all the tests pass, you're done.

Turning the code in

The code will be autograded. Make a zip file containing copies of your **LexicalCategory.cs**, **Sequence.cs**, and **PhrasalCategory.cs** files.

Important: include only those files, not the whole directory. And do not put them inside a directory. They should be at the top level of the zip file.

You can name the zip file anything you like. Upload it to Canvas, and you're done. Congratulations!