Lab 5: Grammar Solver

Due Mar 4 by 11:59pm **Points** 20
Available Feb 17 at 11:59pm - Mar 6 at 11:59pm 17 days

You can turn in your lab using the following link on the codePost website: **https://codepost.io/ (https://codepost.io/)**

As these labs are to be done with a partner(s), please make sure only one person submits the final lab. The other partner(s) should be given the partner link from the submitting partner. This can be found under the "partner" tab after clicking "upload assignment" to submit your assignment in codePost.

We will be using Backus-Naur Form (BNF) grammars for this lab. For more information about these types of grammars you can visit the following link:
https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form. (https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form)

(https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form) This lab will give you practice with recursion and grammars. You will complete a program that reads an input file with a grammar in BNF) and will allow the user to randomly generate elements of the grammar.

You will be given a main program that does the file processing and user interaction. It is called GrammarMain.java. You are to write a class called GrammarSolver that manipulates the grammar. A grammar will be specified as a sequence of Strings, each of which represents the rules for a nonterminal symbol. Each String will be of the form:

<nonterminal symbol>:<rule>|<rule>|<rule>|…|<rule>

Notice that this is the standard BNF format of a non-terminal symbol on the left-hand-side and a series of rules separated by vertical bar characters (“|”) on the right-hand side. If there is only one rule for a particular nonterminal, then there will be no vertical bar characters. Normally BNF productions have the characters “::=” separating the symbol from the rules, so this is a slight variation where the separator is a simple colon (“:”).

There will be exactly one colon per String. The text appearing before the colon is a nonterminal symbol. You may assume that is it not empty and that it does not contain any whitespace. Often we surround nonterminal symbols with the characters “<” and “>”, but this will not always be the case. The text appearing after the colon will be a series of rules separated by vertical bar characters (“|”). Each of

these rules will have a series of tokens (always at least one) separated and potentially surrounded by whitespace. There could be any amount of whitespace surrounding tokens. Any token that appears to the left of a colon in the grammar is considered a nonterminal. All other tokens are considered terminals.

The grammars you will be asked to process will be stored in text files with each line of the file being of the form described above. GrammarMain reads this file into a List<String> and passes the list to the constructor of your GrammarSolver. Your solver has to be able to perform certain tasks, most notably generating random elements of the grammar.

To generate a random instantiation of a non-terminal, you simply pick at random one of its rules and generate whatever that rule tells you to generate. Notice that this is a recursive process. Generating a non-terminal involves picking one of its rules at random and then generating each part of that rule, which might involve more non-terminal symbols to generate for which you pick rules at random and generate each part of those rules, and so on. Depending upon the grammar, this process could continue indefinitely. Any grammar you will be asked to work with will be guaranteed to converge in a finite period of time. Most often this process doesn't go on indefinitely because many rules involve terminals rather than non-terminals. . When you encounter a terminal, you simply include it in the String you are generating. This becomes the base case of the recursive process. Your generating method produces various String objects. **Each String should be compact in the sense that there should be exactly one space between each terminal and there should be no leading or trailing spaces.**

Your class **must** include the methods below although you are allowed to have additional private helper methods if you choose.

GrammarSolver(List<String> grammar)

This method will be passed a grammar as a List of Strings. Your method should store this in a convenient way so that you can later generate parts of the grammar. It should throw an IllegalArgumentException if there are two or more entries in the grammar for the same non-terminal. Your method is not to change the List of Strings.

boolean grammarContains(String symbol)

Returns true if the given symbol is a non-terminal of the grammar; returns false otherwise.

String[] generate(String symbol, int times)

In this method you should use the grammar to randomly generate the given number of occurrences of the given symbol and you should return the result as an array of Strings. For any given non-terminal symbol, each of its rules should be applied with equal probability. It should throw an IllegalArgumentException if the grammar does not contain the given non-terminal symbol or if the number of times is less than 0.

String getSymbols()

This method should return a String representation of the various non-terminal symbols from the grammar as a sorted, comma-separated list enclosed in square brackets, as in “[<np>, <s>, <vp>]”

Case matters when comparing symbols. For example, <S> would not be considered the same as <s>.

The directory crawler program discussed in lecture and provided in the book will serve as a good guide for how to write this program. In that program, the recursive method has a for-each loop. This is perfectly acceptable. Just because we are now learning how to use recursion, we don't want to abandon what we know about loops. If you find that some part of this problem is easily solved with a loop, then go ahead and use one. In the directory crawler, the hard part was writing code to traverse all of the different directories and that's where we used recursion. For your program the hard part is following the grammar to generate different parts of the grammar, so that is the place to use recursion.

You will discover that when writing recursive solutions to problems, we often find ourselves with a public/private pair of methods. You will want to use that approach here. You have been asked to write a public method called generate that will generate an array of Strings. But internally inside your object, you’re going to want to produce these values one String at a time using a recursive method. You should make this internal method private so that it is not visible to the client.

We want you to store the grammar in a particular way. We are making use of the SortedMap<K, V> interface and the implementation TreeMap<K, V>, both in java.util. Maps keep track of key/value pairs. Each key is associated with a particular value. In our case, we want to store something for each non-terminal symbol. So the non-terminal symbols become the keys and the rules become the values. Using this approach, you will find that the getSymbols method can be written quickly because the SortedMap interface includes a method called keySet that returns a set of keys from the map. If you ask for the “toString” of this set, you will get the desired string. It is important to use the SortedMap/TreeMap combination because it keeps the keys in sorted order (notice that getSymbols requires that the non-terminals be listed in sorted order).

Other than the SortedMap/TreeMap restriction, you are allowed to use whatever constructs you want from the Java class libraries, but there are several highly convenient options described below that you are strongly advised to consider.

- The Random class in java.util can be used to generate a random integer by calling its nextInt method.
- The String class has a method called “trim” that will return a new version of the String minus any leading or trailing whitespace.
- One problem you will have to deal with is breaking up strings into various parts. One approach is to use the split method from the String class. It makes use of what are called “regular expressions” and

this can be confusing, but you will find that learning about regular expressions is extremely helpful for computer scientists and computer programmers. Many unix tools, for example, take regular expressions as input.

The regular expressions we want are fairly simple. For example, to split a string on the colon character you simply put the colon inside a String. The split method returns an array of strings, which means we can perform this split by saying:

```
String[] parts = s.split(":");
```

In the case of whitespace, we want to include both spaces and tabs and we want to have one or more of them. This can be accomplished in a regular expression by putting both a space and a tab inside square brackets and putting a plus sign after the brackets which indicates “1 or more of these”:

```
String[] parts = s.split("[ \t]+");
```

One minor issue that comes up with the split above is that if the String you are splitting begins with a whitespace character, you will get an empty String at the front of the resulting array.

In the previous expression the square brackets are used to group two characters together. But they are also useful for special characters. For example, to split on a vertical bar character, we can’t just put the character inside a String as we did with the colon because it has a special meaning in regular expressions. But if we use the bracket notation, this avoids the problem:

```
String[] parts = s.split("[|]");
```

As mentioned above, the various parts of a rule are guaranteed to be separated by whitespace. Otherwise you would have a difficult time separating the parts of a rule. But this means that once you’ve used the spaces to split the rule up, the spaces are gone. That means that when you generate Strings, you will have to include spaces yourself.

Grammar Solver Main File: **GrammarMain.java** ⬇
(https://wcc.instructure.com/courses/2177267/files/186809538/download?download_frd=1)

Sample input file #1: **sentence.txt** ⬇
(https://wcc.instructure.com/courses/2177267/files/186809539/download?download_frd=1)

Sample input file #2: **sentence2.txt** ⬇
(https://wcc.instructure.com/courses/2177267/files/186809537/download?download_frd=1)

Make sure that you're using good style and properly commenting your code. Before submitting your program, check your program against the style guides which can be found through the following link: **Style Guide Links**

You should only be using material we've covered in class so far (Chapters 1 through 16). Using advanced material will result in losing points.