CSE143X: Computer Programming I & II
Programming Assignment #8
due: Monday, 11/20/17, 11:00 pm

This assignment will give you practice with recursion, regular expressions, and grammars and will give you another opportunity to work with maps. You will complete a program that reads an input file with a grammar in Backus-Naur Form (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 nonterminal 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. BNF productions use the characters "::=" to separate the symbol from the rules.

There will be exactly one occurrence of "::=" per string. The text appearing before the "::=" is a nonterminal symbol. You may assume that it is not empty, that it does not contain a vertical bar character, 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 "::=" will be a nonempty 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 "::=" 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 nonterminal, you simply pick one of its rules at random and generate whatever that rule tells you to generate. Notice that this is a recursive process. Generating a nonterminal involves picking one of its rules at random and then generating each part of that rule, which might involve more nonterminal 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 nonterminals. . 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.**

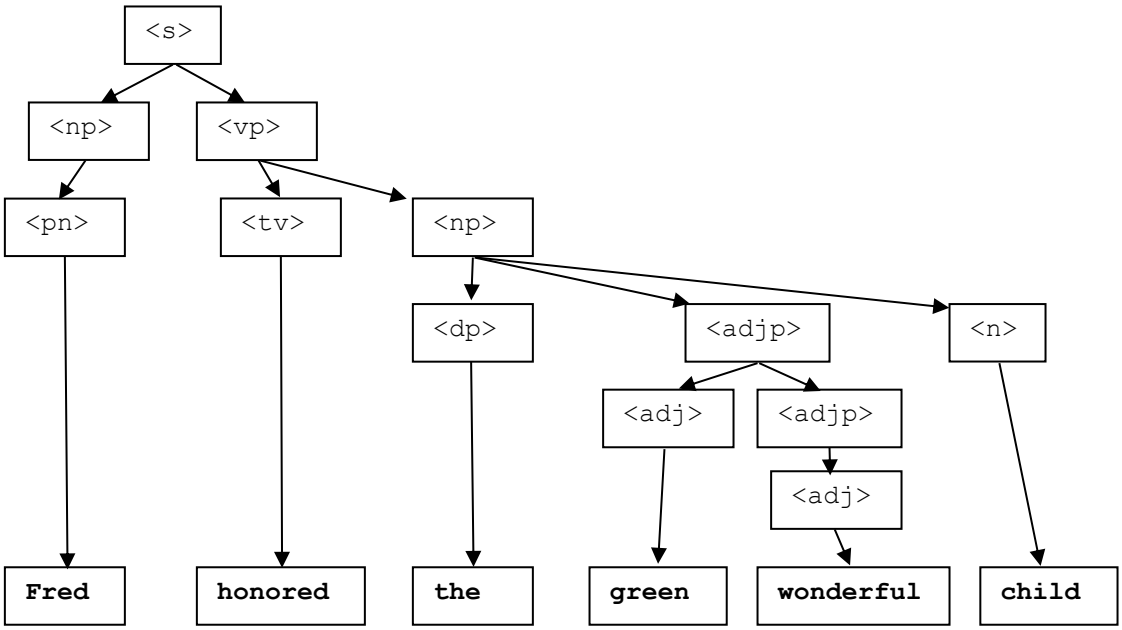
Your class must include the following public methods.

Method	Description
`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 the grammar is empty or if there are two or more entries in the grammar for the same nonterminal. Your method is not to change the list of strings.
`boolean grammarContains(String symbol)`	Returns `true` if the given symbol is a nonterminal 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 nonterminal symbol, each of its rules should be applied with equal probability. It should throw an `IllegalArgumentException` if the grammar does not contain the given nonterminal symbol or if the number of times is less than 0.
`String getSymbols()`	This method should return a `String` representation of the various nonterminal 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 from lecture 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.

Below is an example of a sentence that can be generated from `sentence.txt`. The nonterminal `<s>` can produce the sentence, "Fred honored the green wonderful child", as indicated below:



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 nonterminal symbol. So the nonterminal 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 call `toString` on 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 nonterminals be listed in sorted order).

Below are some specific notes about Java constructs you should be using:

- The `Random` class in `java.util` can be used to generate a random integer by calling its `nextInt` method. You can also use the method `Math.random`.

- The `String` class has a method called `trim` that will return a new version of the string without any leading or trailing whitespace.

- One problem you will have to deal with is breaking up strings into various parts. You should use the `split` method of the `String` class to do so, although you are also allowed to use a string-based `Scanner` if you prefer. The split method 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 `"::="` you simply put the text 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 "::=" 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.

In terms of correctness, your class must provide all of the functionality described above. In terms of style, we will be grading you on use of comments, good variable names, consistent indentation, and otherwise generally good coding style to implement these operations. Remember that you will lose points if you declare variables as data fields that can instead be declared as local variables. You should also avoid extraneous cases (e.g., don't make something into a special case if it doesn't have to be). And you should continue to declare variables, fields, parameters and return types using an interface when possible and applicable.

A collection of files needed for the assignment is included on the web page as `assign8.zip`. You will need to have `GrammarMain.java`, `sentence.txt` and `expression.txt` in the same directory as your `GrammarSolver.java` in order to run `GrammarMain`. The second input file (`expression.txt`) contains extraneous whitespace, including tabs. This input file sometimes generates very long expressions as output.

Sample input file (sentence.txt)

```
<s>::=<np> <vp>
<np>::=<dp> <adjp> <n>|<pn>
<pn>::=John|Jane|Sally|Spot|Fred|Elmo
<adjp>::=<adj>|<adj> <adjp>
<adj>::=big|fat|green|wonderful|faulty|subliminal|pretentious
<dp>::=the|a
<n>::=dog|cat|man|university|father|mother|child|television
<vp>::=<tv> <np>|<iv>
<tv>::=hit|honored|kissed|helped
<iv>::=died|collapsed|laughed|wept
```

Sample input file (expression.txt)

```
E::=    T     |           E               OP     T
T::=   x    |   y   |   42  | 0 | 1 | 92 | ( E ) | F1 ( E ) | - T | F2 ( E , E )
OP::=    +   |      -    |   *       |   %         |        /
F1::=   sin    | cos|    tan  |sqrt   | abs
F2::=max       |min       |   pow
```

Sample execution (user input underlined)

```
Welcome to the cse143 random sentence generator.

What is the name of the grammar file? sentence.txt

Available symbols to generate are:
[<adj>, <adjp>, <dp>, <iv>, <n>, <np>, <pn>, <s>, <tv>, <vp>]
What do you want generated (return to quit)? <dp>
How many do you want me to generate? 5
the
a
a
the
a

Available symbols to generate are:
[<adj>, <adjp>, <dp>, <iv>, <n>, <np>, <pn>, <s>, <tv>, <vp>]
What do you want generated (return to quit)? <np>
How many do you want me to generate? 5
Jane
the wonderful pretentious fat big father
Fred
the wonderful cat
a subliminal pretentious dog

Available symbols to generate are:
[<adj>, <adjp>, <dp>, <iv>, <n>, <np>, <pn>, <s>, <tv>, <vp>]
What do you want generated (return to quit)? <s>
How many do you want me to generate? 20
Jane kissed Spot
the wonderful wonderful pretentious television collapsed
Jane hit Fred
Jane helped a wonderful dog
Elmo helped the fat fat man
a pretentious university helped Elmo
```

```
a wonderful green subliminal father hit Fred
Fred kissed Spot
Spot laughed
the green wonderful father collapsed
the big man helped John
a pretentious green faulty dog collapsed
Jane honored a green subliminal green child
Elmo hit Elmo
a green university died
the pretentious child honored a faulty wonderful subliminal television
Jane died
the faulty dog hit John
Elmo helped Fred
Elmo honored the pretentious big green father

Available symbols to generate are:
[<adj>, <adjp>, <dp>, <iv>, <n>, <np>, <pn>, <s>, <tv>, <vp>]
What do you want generated (return to quit)?
```

Sample execution (user input underlined)

```
Welcome to the cse143 random sentence generator.

What is the name of the grammar file? expression.txt

Available symbols to generate are:
[E, F1, F2, OP, T]
What do you want generated (return to quit)? T
How many do you want me to generate? 5
42
- y
x
x
( ( 1 ) )

Available symbols to generate are:
[E, F1, F2, OP, T]
What do you want generated (return to quit)? E
How many do you want me to generate? 10
x - 1
0
sin ( 1 + 92 + - 1 / 42 )
max ( y , 92 )
42 % 1
- 42
92
1
92
42 - sin ( 1 )

Available symbols to generate are:
[E, F1, F2, OP, T]
What do you want generated (return to quit)?
```