CS 143 Lab 5 Program Specifications

Checkpoint Due 11:59PM Friday Aug 10
Lab Due 11:59PM Monday Aug 13

Problem Overview: This assignment will give you more practice with recursion, and exposure to the important concepts of *regular expressions* and *grammars*. You will also get the opportunity to work with the JCF `Map` and `Set` classes. These outcomes will be achieved when you use a client program that reads an input file with a grammar in *Backus-Naur Form* (BNF) and allows the user to randomly generate elements of the grammar using the `GrammarProcessor` class that you will implement.

The `Lab6Files` folder posted with this assignment contains two client programs `GrammarTest.java` and `GrammerProcessorClient.java`.

- These client files perform the file processing and user interaction for your program.
- `GrammarTest` is a simple tester for the `GrammerProcessor` class that you will write. It may be easier for you to use and modify as needed in the early stages of implementing your `GrammerProcessor` class.
- `GrammerProcessorClient` is more versatile and user-friendly for testing your `GrammerProcessor` class thoroughly in the final stages of development.

BNF Grammars: 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>
```

In the online *Introduction to Programming in Java* 7/e textbook by David J. Eck, Section 9.5.1 presented the following such strings:

```
<sentence> ::= <noun-phrase> | <verb-phrase>
<verb-phrase> ::= <intransitive-verb> | <transitive-verb> <noun-phrase>
```

In practice, the grammar may or may not use <> characters, and for our program, since it will be asking users to type nonterminal symbols from a BNF grammar, the symbols will be very short as in BNF grammar file `sentence1.txt` shown here:

```
<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
```

Here is the BNF grammar contained in `sentence2.txt` (notice it doesn't use <> around its symbols):

```
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 I/O sessions shown below are based on a working `GrammerProcessor` class and the `GrammerProcessorClient`. User input is shown underlined only for easy identification. Since the language components are randomly generated, you should not expect your program to produce the exact same I/O sessions if the user types the exact same input values.

Sample I/O Session for Sentence 1 Grammar:

```
What is the name of the grammar file? sentence1.txt

[<adj>, <adjp>, <dp>, <iv>, <n>, <np>, <pn>, <s>, <tv>, <vp>]
Which symbol do you want? (press <enter> to quit) <s>
How many expressions? 5
the subliminal cat kissed John
Elmo honored Sally
a fat subliminal man hit Jane
the big subliminal fat cat collapsed
Jane collapsed

[<adj>, <adjp>, <dp>, <iv>, <n>, <np>, <pn>, <s>, <tv>, <vp>]
Which symbol do you want? (press <enter> to quit)
```

Sample I/O Session for Sentence 2 Grammar:

```
[E, F1, F2, OP, T]
Which symbol do you want? (press <enter> to quit) E
How many expressions? 5
x
92
min ( x * 0 , - 1 )
1 - x * 0
- - 1 + ( y % x - 42 )

[E, F1, F2, OP, T]
Which symbol do you want? (press <enter> to quit)
```

Regular Expressions and Useful String Methods: Be sure to study the `ReadMary.java` program posted with this assignment. It shows how *regular expressions* can be used to set the delimiters on a `Scanner` object to allow characters other than just whitespace to be ignored while reading tokens from the input stream connected to the `Scanner`.

Notice that each line in the BNF grammars shown above contains exactly one occurrence of `"::="` to separate the nonterminal symbol on the left-hand side from the rules on the right-hand side. Any token that appears to the left of `"::="` in the grammar is considered a nonterminal. All other tokens are considered terminals. The right-hand side will contain the “or” symbol (`"|"`) to separate one rule from another, unless there is only one rule for that nonterminal symbol. Each of the 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.

The `String` class has the following method: `String[] split(String regularExpression)`
It can be used to split a string into substrings delimited by the regular expression. Suppose that `line` is a string with the following value: `"<ex>::=yours|mine|ours"`
Then this statement

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

has the effect of creating `parts[0]` containing `"<ex>"` and `parts[1]` containing `"yours|mine|ours"`.

What if `line` has the value `"yours|mine|ours"`? In order to split the line into substrings delimited by `"|"`, you must use the following syntax:

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

That is, you must enclose the `"|"` symbol inside `[]` because the `"|"` symbol has special meaning in regular expressions. The preceding statement results in `parts[0]` holding `"yours"`, `parts[1]` holding `"mine"`, and `parts[2]` holding `"ours"`.

Lastly, suppose that `line` has the following value: `" to    be   or   not    to be"`

There are whitespace characters separating the symbols in each rule from each other. To split the string into substrings delimited by whitespace, we would use the following statement:

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

This says there are one or more spaces or tabs to ignore between splits. The only issue with the statement above is that if `line` begins with a whitespace character, `parts[0]` will contain the empty string `""`.

So another `String` class method that may be useful for this assignment is the following:

`String trim()` can be used to create a string that has leading and trailing spaces removed.

```
String st = s.trim();    //st holds trimmed version of s; s is unchanged.
```

Java Collections Framework: You will be using `Set`, `Map` and possibly `Iterator` objects in this assignment. Those objects will handle all your data storage needs; you just need to know how to create and access them. Specifically, you will be storing grammar rules in the following type of object:

`TreeMap<String, String[]>` object.

Think about it… Draw a picture of sample grammar rules stored in a map so that you understand why this choice of data structure makes sense for this lab. Make sure that you understand what the `get()`, `put()`, `containsKey()`, `keyset()` map methods listed in Table 11.5 on page 726 (704 in 2/e) do. Since the `keyset()` method returns a `Set` object, be sure that you understand how to traverse a `Set` using either an `Interator` object, or a `for`-each loop.

GrammarProcessor Specifications: `GrammarTest` and `GrammerProcessorClient` will read a BNF grammar from a text file and store each line from the file into a `List<String>` that is passed to your `GrammarProcessor` constructor. Your constructor will store the grammar in a manner that makes it convenient to generate random elements of the grammar, as shown in the specification table below.

GrammarProcessor Class	
Class/Instance Data	**Description**
`private Map<String,String[]> rules = new TreeMap<String,String[]>();`	The *nonterminals* of the grammar will be the key fields for the map. The rules for each nonterminal will be stored in the `String[]` array associated with the nonterminal key value. Note that since we are using a `TreeMap` implementation of the `Map` interface, the keys will be stored in sorted order.
Public methods	**Description**
`GrammarProcessor(List<String> grammar)`	Each entry in the `grammar` parameter holds a line from a BNF grammar file. This constructor will store the `grammar` data in the `rules` map appropriately. It should throw an `IllegalArgumentException` if the `grammar` is `null`, empty or contains more than one entry for the same nonterminal. Note that case matters, so `<S>` and `<s>` are not the same. The constructor should not change the contents of `grammar` in any way. **You are strongly encouraged to display the contents of the `rules` map during development so that you are absolutely certain you have the grammar correctly stored in the map before you code the `generate()` methods described below.**
`boolean grammarContains(String symbol)`	Returns true if the symbol is a nonterminal of the grammar stored in `rules`, or false otherwise.
`String getSymbols()`	Returns a String representation of all the nonterminal symbols of the grammar. This should be a sorted, comma-separated listing enclosed within square brackets, as in `"[<np>, <s>, <vp>]"` Hint: Apply the `toString()` method to `keyset()` for our map.
`String generate(String symbol)`	Use the grammar in the `rules` map to randomly generate an occurrence of the given `symbol`. Throw an `IllegalArgumentException` if the grammar does not contain the given nonterminal `symbol`. *Call a private recursive helper method appropriately.*

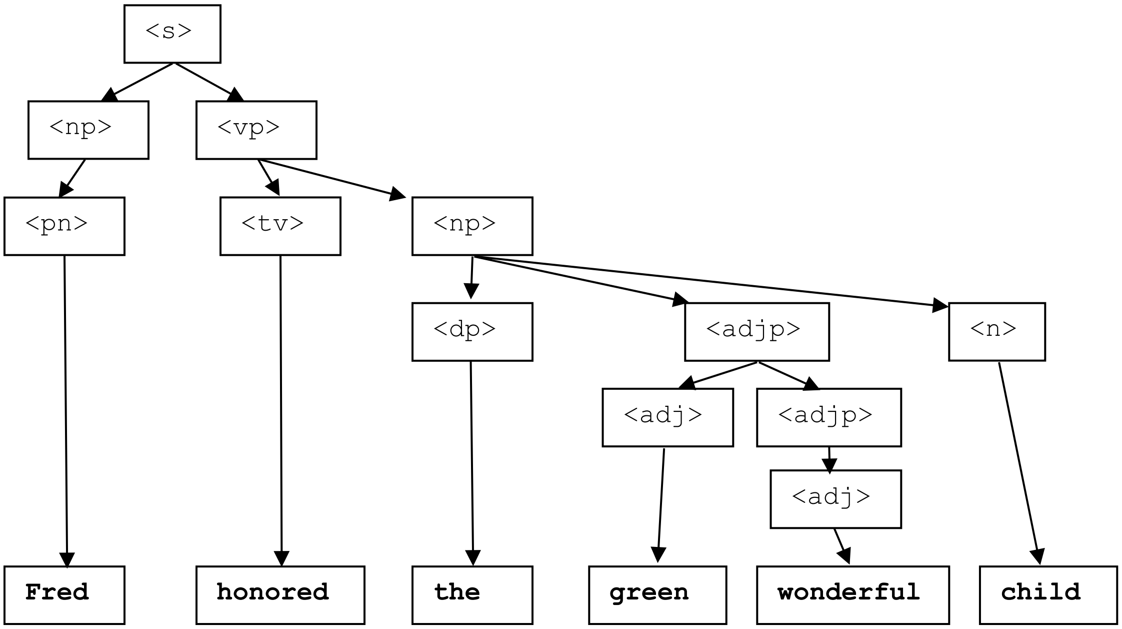
`GrammarProcessor` Specs continued below.

GrammarProcessor Class	
Public methods	**Description**

`String[] generate(String symbol,`
`int times)`
	This method does the same thing as the preceding method, except that it returns multiple `(times)` randomly generated occurrences of the given `symbol` in a String array. It should throw an `IllegalArgumentException` under the same circumstances as above, and if `times` is less than 0. *Call a private recursive helper method appropriately.*

In addition to the `public generate()` methods described above, you will write `private` helper methods that perform the actual recursion. Typically, the private method has the same name as the public, but a different parameter list. To generate a random instantiation of a nonterminal, you will pick at random one of its rules and generate whatever that rule tells you to generate. This is a recursive process because a rule might lead to another nonterminal, which will lead to another nonterminal, and so on. When you finally encounter a terminal, you simply include it in the String you are generating. This becomes the base case of the recursive process. ***See the `FileCrawler.java` program presented in Chapter 12 and posted in the `Lab5Files` folder. The folder also contains a dummy `crawl_dir` folder that you can ask the `FileCrawler` to display for you. Your public/private `generate()` method pairs will be very similar to the public/private `crawl()` method pairs.*** Note that the recursive `crawl()` method contains a loop. There is no rule against having a loop in a recursive method.

The grammar in `sentence1.txt` can produce the sentence (<s>), “Fred honored the green wonderful child”, as indicated in the following diagram:



Be sure to follow the good programming style specifications posted in the Quick Links module on the Canvas website.