CSE 143: Computer Programming II — Summer 2017

HW4: Grammar (due Tuesday, July 25 2017 11:30pm)

This assignment focuses on recursive programming, regular expressions, and grammars. It will also give you an opportunity to work with maps. Turn in the following files using the link on the course website:

- `Grammar.java` – A class that allows for the generation of grammars based on given rules.
- `grammar.txt` – A text file containing your custom-made grammar

You will need the support files `GrammarMain.java`, and `sentence.txt`; place these in the same folder as your program or project. The code you submit must work properly with the unmodified versions of the provided files.

Languages, Grammars, and BNF

The main goal of this assignment is to generate random valid sentences given a set of rules. To explain how to do this, we first need to define a few ideas.

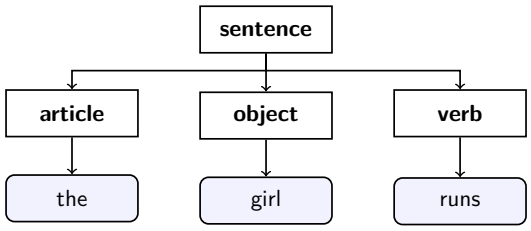
Formal Languages

A *formal language* is a set of words and symbols along with a set of rules defining how those symbols may be used together. For example, "A boy threw the ball." is a valid sentence, but "A threw boy ball the" makes no sense, because the words were put together in an invalid way.
In this assignment, we will use a special syntax called Backus-Naur Form (*BNF*) to describe what constitutes a valid sentence.

Grammars

A *grammar* is a way of describing the syntax and symbols of a formal language. Grammars have two types of "symbols" (e.g., words, phrases, sentences): *terminals* and *non-terminals*. A *terminal symbol* is a fundamental word of the language. For example, in English, any single word would be a terminal. When constructing sentences, we want to put words together in grammatically correct ways using sentences, noun phrases, objects, etc. You can think of non-terminals as specific combinations of choices of terminals. For example, consider the following simple language:

- Terminals: the, a, boy, girl, runs, walks
- Non-terminals:
  - A **sentence** is "article **and** object **and** verb"
  - An **article** is "the **or** a".
  - An **object** is "boy **or** girl".
  - A **verb** is "runs **or** walks".



This language has the following sentences:

"the boy runs"	"the boy walks"	"a boy runs"	"a boy walks"
"the girl runs"	"the girl walks"	"a girl runs"	"a girl walks"

Backus-Naur Form (BNF)

BNF is a specific format for specifying grammars. Each line of a BNF looks like the following:

```
nonterminal ::= rule | rule | ... | rule
```

Each "rule" is either a terminal or a non-terminal. The grammar we specified above would look like the following in BNF:

```
sentence ::= article object verb
article  ::= the | a
object   ::= boy | girl
verb     ::= runs | walks
```

Notice that *sentence* has multiple non-terminals put together in one choice, whereas the others each consist of one of multiple choices.
We guarantee the following formatting rules of BNFs you will be given:

- Each line will contain *exactly one occurrence* of `::=` which is the separator between the name of the non-terminal and the choices.
- A pipe (`|`) will separate each choice for the non-terminal. If there is only one rule (like with `sentence` above), there will be no pipe on that line.
- Whitespace separates tokens but doesn't haven any special meaning. There will be at least one whitespace character between each part of a single rule. Extra whitespace should be removed.
- Case matters when comparing symbols. For example, `<S>` would not be considered the same non-terminal as `<s>`.
- If a symbol on the right never appears on the left of a `::=`, it should be considered a *terminal*.

Grammar

In this assignment you will complete a program that reads an input file with a grammar in Backus-Naur Form and allows the user to randomly generate elements of the grammar. You will use **recursion** to implement the core of your algorithm.

You are given a client program `GrammarMain.java` that does the file processing and user interaction. You are to write a class called `Grammar` that manipulates a grammar. `GrammarMain` reads a BNF grammar input text file and passes its entire contents to you as a list of strings. You must break each string from the list into symbols and rules so that it can generate random elements of the grammar as output.

`Grammar` should have the following constructor:

```
public Grammar(List<String> rules)
```

This constructor should initialize a new grammar over the given BNF grammar rules where each rule corresponds to one line of text. You should use *regular expressions* (see below) to break apart the rules and store them into a `Map` so that you can look up parts of the grammar efficiently later.

You should not modify the list passed in. You should throw an `IllegalArgumentException` if the list is `null`, empty, or if there are two or more entries in the grammar for the same non-terminal.

`Grammar` should also implement the following methods:

`public boolean` **`isNonTerminal`**`(String symbol)`

This method should return `true` if the given symbol is a *non-terminal* in the grammar and `false` otherwise. You should throw an `IllegalArgumentException` if the string is null or has length 0.

For example, for the grammar above, `isNonTerminal("sentence")` would return `true` and `isNonTerminal("foo")` or `isNonTerminal("boy")` ("boy" is a terminal in the language) would return `false`.

`public String` **`toString`**`()`

This method should return a string representing all non-terminal symbols of your grammar in alphabetical order. You will want to use the `keySet` of your map.

For example, calling `toString()` for the previous grammar would give: "`[article, object, sentence, verb]`".

`public String[]` **`getRandom`**`(String symbol, int times)`

This method should generate *times* random occurrences of the given *symbol* and return them as a `String[]`. **Each string generated 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.**

If *times* is negative, you should throw an `IllegalArgumentException`. If the `String` argument passed *is not* a non-terminal in your grammar (or it is `null`), you should throw an `IllegalArgumentException`.

When generating a non-terminal symbol in your grammar, each of its rules should be applied with equal probability. Use the `Random` class in `java.util` to help you make random choices between rules.

Implementation Details

`Grammar` Constructor

We want you to store the grammar in a particular way using a `Map`. The *keys* of your `Map` will be the *non-terminals* and the *values* will be the rules for a specific non-terminal.

Note that you will want to store the keys in your map in a particular order (for your `toString()` method), and you will want to store your grammar rules for each non-terminal in a way that is convenient for the operations needed in this assignment.

`getRandom` Algorithm

In the `getRandom` method, the main goal is to generate a random occurrence of a given non-terminal NT. You should use the following recursive algorithm:

> Choose a random expansion rule R for the non-terminal NT. For each of the symbols in the rule R, generate a random occurrence of that symbol. Note that your base case will be when the symbols in the rule you chose are *not* non-terminals. Meanwhile, you will want to recurse for any non-terminals you need to generate.

It is perfectly okay to have a loop inside your recursion. In fact, your code will be better in this assignment if you include one. You should look back to the later recursion lectures for a reminder if you can't remember how this works.

Testing Your Solution

There is an Output Comparison Tool for this assignment but it is only so helpful since the output is random. We are providing another tool that is linked on the section for this assignment to check the output of your `getRandom` method to make sure it is producing valid output.

You can test your whitespace of your `getRandom` by using some non-whitespace character (i.e. ~) instead of spaces and inspecting the output visually.

Splitting Strings

In this assignment, it will be useful to know how to *split* strings apart in Java. In particular, you will need to split the various options for rules on the | character, and then, you will need to split the pieces of a rule apart by spaces.

To do this, you should use **String's split method** which takes in a `String` delimiter (e.g. "what to split by") argument and returns your original large `String` as an array of smaller `Strings`.

The delimiter `String` passed to `split` is called a *regular expression*, which are strings that use a particular syntax to indicate patterns of text. A regular expression is a `String` that "matches" certain sequences. For instance, "abc" is a regular expression that matches "a followed by b followed by c". The following regular expressions will be useful to you for this assignment:

- **Splitting Non-Terminals from Rules.** Given a `String`, `line`, to split `line` based on where `::=` occurs, you could use the regular expression "`::=`" (since you are looking for these *literal* characters). For example:

```
1 String line = "example::=foo bar |baz";
2 String[] pieces = line.split("::="); // ["example", "foo bar |baz"]
```

- **Splitting Different Rules.** Given a `String`, `rules`, to split `rules` based on where the | character is, it looks similar to the above, *except*, in regular expressions, | is a special character. So, we must escape it (just like `\n` or `\t`). So, the regular expression is "`\\|`". (Note that we need two slashes because slashes themselves must be escaped in `Strings`.) For example:

```
1 String rules = "foo bar|baz |quux mumble";
2 String[] pieces = rules.split("\\|"); // ["foo bar", "baz ", "quux mumble"]
```

- **Splitting Apart a Single Rule.** Given a `String`, `rule`, to split `rule` based on whitespace, we must look for "at least one whitespace". We can use `\\s` to indicate "a single whitespace of any kind: `\t`, space, etc. And by adding + afterwards, the regular expression is interpreted as "one or more of whitespace". For example:

```
1 String rule = "the quick brown    fox";
2 String[] pieces = rule.split("\\s+"); // ["the", "quick", "brown", "fox"]
```

Removing Whitespace from the Beginning and the End of a String

One minor issue that comes up with splitting on whitespace as 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.

Given a `String`, `str`, we can create a new `String` that omits all leading and trailing whitespace removed:

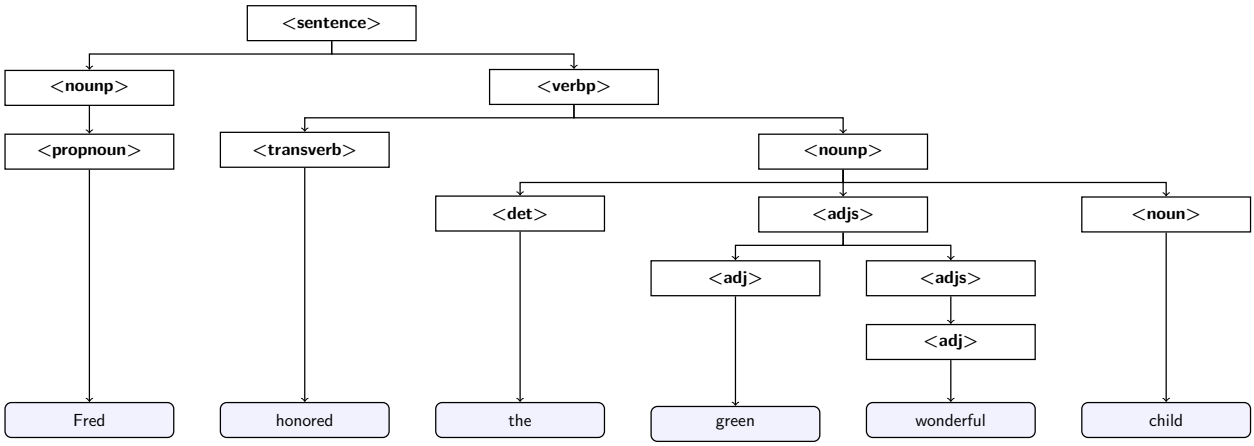
```
1 String str = " lots of spaces \t";
2 String trimmedString = str.trim(); // "lots of spaces"
```

Full Example Walk-Through

Complex BNF (sentence.txt)

```
<sentence>::=<nounp> <verbp>
<nounp>::=<det> <adjs> <noun>|<propnoun>
<propnoun>::=John|Jane|Sally|Spot|Fred|Elmo
<adjs>::=<adj>|<adj> <adjs>
<adj>::=big|green|wonderful|faulty|subliminal|pretentious
<det>::=the|a
<noun>::=dog|cat|man|university|father|mother|child|television
<verbp>::=<transverb> <nounp>|<intransverb>
<transverb>::=hit|honored|kissed|helped
<intransverb>::=died|collapsed|laughed|wept
```

Example Random Sentence



Example Execution

```
>> Welcome to the CSE 143 Random Sentence Generator!
>>
>> What is the name of the grammar file? sentence.txt
>>
>> Available non-terminals are:
>> [<adj>, <adjs>, <det>, <intransverb>, <noun>, <nounp>, <propnoun>, <sentence>, <transverb>, <verbp>]
>> Which non-terminal do you want to generate (Enter to quit)? <sentence>
>> How many do you want me to generate? 5
>>
>> Sally hit Jane
>> Spot hit John
>> Jane died
>> the green mother wept
>> the subliminal green man laughed
>>
>> Available non-terminals are:
>> [<adj>, <adjs>, <det>, <intransverb>, <noun>, <nounp>, <propnoun>, <sentence>, <transverb>, <verbp>]
>> Which non-terminal do you want to generate (Enter to quit)?
```

Development Strategy

The hardest method is `getRandom`, so write it last. The hard part of `getRandom` is following the grammar rules to generate different parts of the grammar, so that is the place to use recursion. The directory crawler program from lecture is a good thing to study if you are stuck.

If your recursive method has a bug, try putting **a debug `println`** that prints your parameter values to see the calls being made.

Creative Aspect (grammar.txt)

You will also submit a file `grammar.txt` that contains a valid BNF grammar that can be used as input. For full credit, the file should be in valid BNF format, contain at least 5 non-terminals, and should be your own work (do more than just changing the terminal words in `sentence.txt`, for example).

Style Guidelines and Grading

Part of your grade will come from appropriately utilizing recursion to implement your algorithm as described previously. We will also grade on the elegance of your recursive algorithm; don't create special cases in your recursive code if they are not necessary. Redundancy is another major grading focus; you should avoid repeated logic as much as possible. Your class may have other methods besides those specified, but any other methods you add should be private.

Avoid Redundancy

Create "helper" method(s) to capture repeated code. As long as all extra methods you create are `private` (so outside code cannot call them), you can have additional methods in your class beyond those specified here. If you find that multiple methods in your class do similar things, you should create helper method(s) to capture the common code.

Generic Structures

You should always use generic structures. If you make a mistake in specifying type parameters, the Java compiler may warn you that you have "unchecked or unsafe operations" in your program. If you use jGRASP, you may want to change your settings to see which line the warning refers to. Go to `Settings/Compiler Settings/Workspace/Flags/Args` and then uncheck the box next to "`Compile`" and type in: `-Xlint:unchecked`

Data Fields

Properly encapsulate your objects by making data your fields `private`. Avoid unnecessary fields; use fields to store important data of your objects but not to store temporary values only used in one place. Fields should always be initialized inside a constructor or method, never at declaration.

Java Style Guidelines

Appropriately use control structures like loops and if/else statements. Avoid redundancy using techniques such as methods, loops, and factoring common code out of if/else statements. Properly use indentation, good variable names, and types. Do not have any lines of code longer than 100 characters.

Commenting

You should comment your code with a heading at the top of your class with your name, section, and a description of the overall program. All method headers should be commented as well as all complex sections of code. Make sure you describe complex methods inside methods. Comments should explain each method's behavior, parameters, return values, and assumptions made by your code, as appropriate. The `ArrayIntList` class from lecture provides a good example of the kind of documentation we expect you to include. You do not have to use the pre/post format, but you must include the equivalent information–including the type of exception thrown if a precondition is violated. Write descriptive comments that explain error cases, and details of the behavior that would be important to the client. Your comments should be written in your own words and not taken verbatim from this document.