CS145 – PROGRAMMING ASSIGNMENT

GRAMMER SOLVER

OVERVIEW

This program focuses on programming with recursion.

INSTRUCTIONS

Your deliverable will be to turn in a file named `GrammarSolver.java` to the course web site.

You will need support files `GrammarMain.java`, `sentence.txt`, and other input files from the course web site; place them in the same folder as your project.

LANGUAGES AND GRAMMERS

A formal language is a set of words and/or symbols along with a set of rules, collectively called the syntax of the language, defining how those symbols may be used together. A grammar is a way of describing the syntax and symbols of a formal language. Many language grammars can be described in a common format called Backus-Naur Form (BNF).

Some symbols in a grammar are called terminals because they represent fundamental words of the language. A terminal in the English language might be the word "boy" or "run" or "Jessica". Other symbols of the grammar are called nonterminals and represent high-level parts of the language syntax, such as a noun phrase or a sentence. Every non-terminal consists of one or more terminals; for example, the verb phrase "throw a ball" consists of three terminal words.

The BNF description of a language consists of a set of derivation rules, where each rule names a symbol and the legal transformations that can be performed between that symbol and other constructs in the language. For example, a BNF grammar for the English language might state that a sentence consists of a noun phrase and a verb phrase, and that a noun phrase can consist of an adjective followed by a noun or just a noun. Rules can be described recursively (in terms of themselves). For example, a noun phrase might consist of an adjective followed by another noun phrase.

A BNF grammar is specified as an input file containing one or more rules, each on its own line, of the form:

```
non-terminal ::=  rule | rule | rule | ... | rule
```

A `::=` (colon colon equals) separator divides the non-terminal from its expansion rules. There will be exactly one `::=` per line. A `|` (pipe) separates each rule; if there is only one rule for a given non-terminal, there will be no pipe characters.

The following is a valid example BNF input file describing a small subset of the English language. Non-terminal names such as `<s>`, `<np>` and `<tv>` are short for linguistic elements such as sentences, noun phrases, and transitive verbs.

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

The language described by this grammar can represent sentences such as "The fat university laughed" and "Elmo kissed a green pretentious television". This grammar cannot describe the sentence "Stuart kissed the teacher" because the words "Stuart" and "teacher" are not part of the grammar. The grammar also cannot describe "fat John collapsed Spot" because there are no rules that permit an adjective before the proper noun "John", nor an object after intransitive verb "collapsed".

Though the non-terminals in the previous language are surrounded by `<  >`, this is not required. By definition any token that ever appears on the left side of the `::=` of any line is considered a non-terminal, and any token that appears only on the right-hand side of `::=` in any line(s) is considered a terminal. Each line's non-terminal will be a non-empty string that does not contain any whitespace. Each rule might have surrounding spaces around it, which you will need to trim. There also might be more than one space between parts of a rule, such as between `<tv>` and `<np>` below. For example, the following would be a legal equivalent of the last three lines of the previous grammar:

```
<vp> ::= tv <np> | iv
tv   ::= hit | honored | kissed| helped
iv   ::= died | collapsed | laughed | wept
```

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.

Your program should exactly reproduce the format and general behavior demonstrated in this log. Because your program involves randomness (described below), the exact phrases generated may not match.

SAMPLE EXECUTION

```
Welcome to the 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 to generate (Enter to quit)? <dp>
How many do you want me to generate? 3

the
the
a

Available symbols to generate are:
[<adj>, <adjp>, <dp>, <iv>, <n>, <np>, <pn>, <s>, <tv>, <vp>]

What do you want to generate (Enter to quit)? <np>
How many do you want me to generate? 5

a wonderful father
the faulty man
Spot
the subliminal university
Sally

Available symbols to generate are:
[<adj>, <adjp>, <dp>, <iv>, <n>, <np>, <pn>, <s>, <tv>, <vp>]

What do you want to generate (Enter to quit)? <s>
How many do you want me to generate? 10
```

```
a pretentious dog hit Elmo
a green green big dog honored Fred
the big child collapsed
a subliminal dog kissed the subliminal television
Sally laughed
Fred wept
Fred died
the pretentious fat subliminal mother wept
Elmo honored a faulty television
Elmo honored Elmo

Available symbols to generate are:
[<adj>, <adjp>, <dp>, <iv>, <n>, <np>, <pn>, <s>, <tv>, <vp>]

What do you want to generate (Enter to quit)?
```

RECURSIVE ALGORITHM

You can generate elements of a grammar using a recursive algorithm. To generate a random occurrence of a symbol S:

- If S is a terminal symbol, there is nothing to do.
- If S is a non-terminal symbol, choose a random expansion rule R for S. For each of the symbols in the rule R, generate a random occurrence of that symbol.

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 :
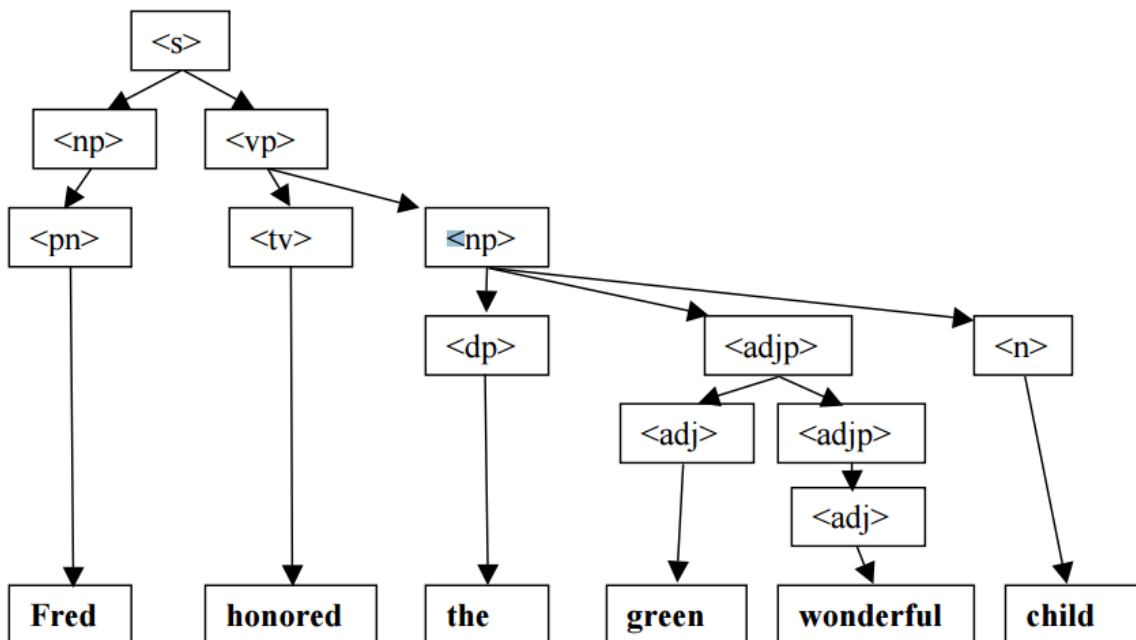
To generate a non-terminal, pick one of its rules at random and then generate each part of that rule, which might involve more non-terminals to recursively generate. For each part, pick a rule at random and generate each of its part, etc. When you encounter a terminal, simply include it in your string. This becomes a base case of the process.

You will be given a client program GrammarMain.java that does the file processing and user interaction. You are to write a class called GrammarSolver that manipulates a grammar. GrammarMain reads a BNF grammar input text file and passes its entire contents to you as a list of strings. For example, if your program were to examine the grammar on the previous page, your object would be passed a 10-element list of the entire contents of that grammar file. Your solver must break each string from the list into symbols and rules so that it can generate random elements of the grammar as output.

NECESSARY METHODS

PUBLIC GRAMMARSOLVER(LIST<STRING> RULES)

In this constructor you should initialize a new grammar solver over the given BNF grammar rules, where each rule corresponds to one line of text as shown in the file on the previous page. Your constructor should break apart the rules and store them into a **Map** so that you can later look up parts of the grammar efficiently. Do not modify the list passed.

You should throw an `IllegalArgumentException` if the list is null, or empty (size 0). You should also throw an `IllegalArgumentException` if the grammar contains more than one line for the same non-terminal. For example, if two lines both specified rules for symbol "<s>" , this would be illegal and should result in the exception being thrown.

PUBLIC BOOLEAN CONTAINS(STRING SYMBOL)

In this method you should return `true` if the given symbol is a non-terminal in the grammar and false otherwise. For example, when using the grammar described previously, return true for a call of contains("`<s>`") and false for a call of contains("`<foo>`") or contains("`green`") ("`green`" is a terminal in the language).

You should throw an `IllegalArgumentException` if the string is null or has a length of 0.

PUBLIC SET<STRING> GETSYMBOLS()

In this method you should return all non-terminal symbols of your grammar as a sorted set of strings. This is the keyset of your map. For example, when using the previous grammar, getSymbols() would return a set containing the ten elements

```
["<adj>", "<adjp>", "<dp>", "<iv>", "<n>", "<np>", "<pn>", "<s>",
"<tv>", "<vp>"].
```

In this method you should use the grammar to generate a random occurrence of the given symbol and you should return it as a `String`. For example, when using the grammar described on the previous pages, a call of `generate("<np>")` might potentially return the string, `"the green wonderful child"`. If the string passed is not a non-terminal in your grammar, you should assume that it is a terminal symbol and simply return it. For example, a call of `generate("green")` should return "green". (Note there is not a space before/after "green".)

For any nonterminal symbol, 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.

You should throw an `IllegalArgumentException` if the string is null or has a length of 0.

DEVELOPMENTS STRATEGY AND HINTS

The hardest method is generate(), so write it last.

Very important! Look up the string method String.trim() and make **heavy** use of it. It will solve some problems with your output as you get deeper into the program.

For your program the hard part is following the grammar rules to generate different parts of the grammar, so that is the place to use recursion.

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

For this program you should store the contents of the grammar into a Map. As you know, maps keep track of key/value pairs, where each key is associated with a particular value. In our case, we want to store information about each nonterminal symbol. So the non-terminal symbols become keys and their rules become values.

Notice that the getSymbols method requires that the non-terminals be listed in sorted order, which may affect what kind of map you use.

Other than the Map requirement, you are allowed to use whatever constructs you want from the Java class libraries.

STRING SPLITTING

One problem you will have to deal with early in this program is breaking strings into various parts.

Although there are several ways to do this, for this assignment, I want you to focus on using the following technique.

We know how a scanner can read a string and separate the tokens one by one, but it is possible to alter a scanner to separate a string by something other than whitespace. To do this you need to alter the scanner to focus on a new "delimiter" when parsing a file. To accomplish this, we can use the `Scanner.useDelimiter()` method.

For example, to split a string by `::=` characters you do the following.

- Create the scanner like normal.

```
String s1 = "example ::= 1 | 2 | 3 | 4 5 | 6 7";
Scanner lineScanner = new Scanner(s1);
```

- Alter the scanner to use the new delimiter:

```
lineScanner.useDelimiter("::=");
```

- Now pull out the first "token" which will be everything in front of the `::=`.

```
String front = lineScanner.next();
```

- Now pull out the second "token" which will be everything in back of the `::=`.

```
String back  = lineScanner.next();
```

You now have spilt your string using a scanner with an alternate delimiter. Now you will need to do it again for the possible rules, and then again for the parts within the individual rules. But you can play with that.

Note: For this assignment, you **must** use the delimiter approach shown above, use of any other spitting technique, such as Strings.split() or converting to an array is not valid.

STYLE GUIDELINES AND GRADING:

Part of your grade will come from appropriately utilizing recursion to implement your algorithm as described previously.

I 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.

You should follow good general style guidelines such as: making fields private and avoiding unnecessary fields; declaring collection variables using interface types; appropriately using control structures like loops and if/else; properly using indentation, good variable names and types; and not having any lines of code longer than 100 characters.

Comment your code descriptively in your own words at the top of your class, each method, and on complex sections of your code. Comments should explain each method's behavior, parameters, return, pre/post-conditions, and exceptions.

For reference, a possible solution is than 100 lines long including blank lines (75 "substantive" lines).