CSE 143, Summer 2011
Programming Assignment #4: Grammar Solver
Due Thursday, July 21, 2011, 11:00 PM

This program focuses on programming with recursion. Turn in files named `GrammarSolver.java` and `grammar.txt` using the links on the Homework section of the course web site. You will need support files `GrammarMain.java`, `sentence.txt`, and other input files from the Homework section of the course web site; place them in the same folder as your project.

Languages and Grammars:

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 *non-terminals* 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
```

Sample input file `sentence.txt`

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
```

Program Description:

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 `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 was to examine the grammar on the previous page, your object would be passed a 10-element list of strings of the entire contents of that grammar file. Your solver must break that list into its symbols and rules so that it can generate random elements of the grammar as output.

Your program should exactly reproduce the format and general behavior demonstrated in this log, although you may not exactly recreate this scenario because of the shuffling of the names that your code performs.

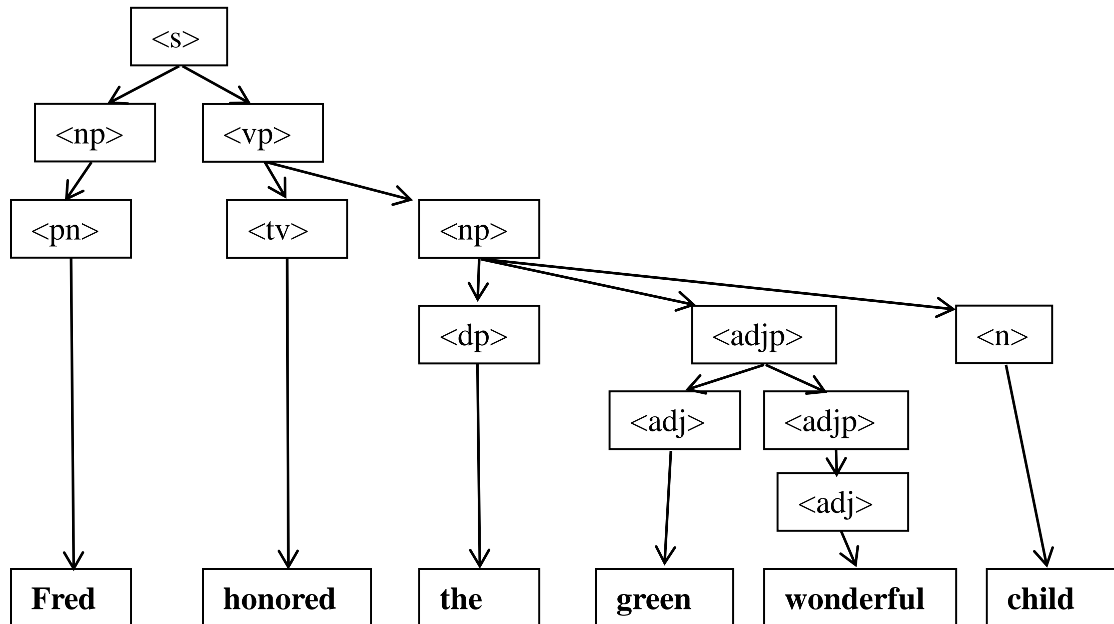
```
Welcome to the CSE 143 random sentence generator!
What is the name of the grammar file? sentence

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.

For example, the grammar on the previous page could be used to randomly generate a `<s>` non-terminal for the sentence, "Fred honored the green wonderful child", as shown in the diagram on the next page:



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-terminals to recursively generate. For each of these you pick rules at random and generate each part, etc. When you encounter a terminal, simply include it in your string. This becomes a base case of the process.

Required 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, you would 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>"]`.

`public String generate(String symbol)`

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`. If the string passed is a non-terminal in your grammar, you should use the grammar's rules to recursively expand that symbol fully into a sequence of terminals. 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"`.)

You may want to look up the methods of 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.

Development Strategies and Hints:

The hardest method is `generate`, so write it last. The directory crawler program from lecture is a good guide for how to write this program. In that program, the recursive method has a for-each loop. This is perfectly acceptable; if you find that part of this problem is easily solved with a loop, 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 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. Or use the debugger interface in jGrasp to place a breakpoint at the beginning of the method and look at the parameters and other variables each time it is called.

For this program **you must 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 non-terminal 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.

[Big hint: There are several implementations of Maps in the Java class libraries. You may find it very useful to look at the SortedMap interface and TreeMap implementation.]

One problem you will have to deal with early in this program is breaking strings into various parts. There are several ways to do this, but we strongly recommend that you use the **`String`'s `split` method**. The `split` method breaks a large string into an array of smaller string tokens; it accepts a *delimiter* string parameter and looks for that delimiter as the divider between tokens. The delimiter strings passed to `split` are called *regular expressions*, which are strings that use a particular syntax to indicate patterns of text. They can be confusing, but learning about regular expressions is helpful for computer scientists and programmers. Many Unix/Linux tools, for example, use regular expressions as input.

To split a string by `::=` characters you simply pass those characters to `split`. To split by whitespace, we want our delimiter to be a sequence of one or more spaces and/or tabs. This can be accomplished by putting a space and a tab inside `[ ]` brackets and putting a + plus sign after the brackets to indicate "1 or more". To split on a pipe character, we can't just pass the pipe character as a String as we did with the `::=` because `|` has a special meaning in regular expressions. So we must enclose it in `[ ]` brackets as well. The following examples show these **regular expressions**:

```
String s1 = "example::=foo bar |baz";
String[] parts1 = s1.split("::=");           // ["example", "foo bar |baz"]
```

```
String s2 = "the quick    brown        fox";
String[] parts2 = s2.split("[ \t]+");        // ["the", "quick", "brown", "fox"]
```

```
String s3 = "foo bar|baz  |quux  mumble";
String[] parts3 = s3.split("[|]");           // ["foo bar", "baz  ", "quux  mumble"]
```

If the string you split begins with a space, you will get an empty string at the front of your array, so use the `String` **`trim` method** as needed. Also, the parts of a rule will be separated by whitespace, but once you've split the rule by spaces, all spaces are gone. If you want spaces between words when generating strings to return, you must include these yourself.

Creative Aspect (`grammar.txt`):

Along with your program, 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). This will be worth a small portion of your grade.

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`.

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, our solution is around 75 lines long including comments and blank lines (35 "substantive" lines).