CSE 143: Computer Programming II — Winter 2021

Take-home Assessment 5: Grammar Solver *due February 11, 2021 11:59pm*

This assignment will assess your mastery of the following objectives:

- Implement a well-designed Java class to meet a given specification.
- Implement recursive methods to solve a naturally-recursive problem.
- Implement a public-private recursive pair.
- Choose an appropriate data structure to represent specified data.
- Follow prescribed conventions for code quality, documentation, and readability.

Overview: Languages, Grammars, and BNF

In this assessment, you will write a class `GrammarSolver` that will be able to generate random sentences (or other output) from a set of specially-formatted rules. These rules are called a *grammar* and are used to define a *language*. Our grammars will be written in *Backus-Naur Form (BNF)*.

Formal Languages

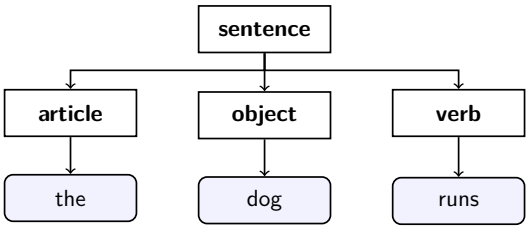
A *formal language* is a set of words and symbols along with a set of rules defining how those symbols may be used together. These rules dictate what are considered valid constructions in the defined language. For example, in English, "A boy threw the ball." is a valid sentence, but "A threw boy ball the" is not, despite consisting of the same words, because the words are put together in an invalid way.

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* is a fundamental word or symbol in the language. For example, in English, any single word would be considered a terminal. A *non-terminal* is a symbol that is used to define specific groups of symbols that may be used in the language. In a grammar for English, we might have non-terminals such as "adjective," "noun phrase," and "sentence" to name a few.

For example, consider the following simple language:

- *Terminals:* the, a, cat, dog, runs, walks
- *Non-terminals:*
  - **sentence**: "article **and** object **and** verb"
  - **article**: "the **or** a".
  - **object**: "cat **or** dog".
  - **verb**: "runs **or** walks".



This language allows the following sentences:

"the cat runs"	"the cat walks"	"a cat runs"	"a cat walks"
"the dog runs"	"the dog walks"	"a dog runs"	"a dog walks"

Backus-Naur Form (BNF)

Backus-Naur Form (BNF) is a specific format for specifying grammars. Each line of BNF looks like the following:

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

Each "rule" is some sequence of terminals or non-terminals separated by whitespace. The '|' character separates different possible rules for the same non-terminal. For example, the grammar specified above written in BNF would look like:

```
sentence::=article object verb
article::=the|a
object::=cat|dog
verb::=runs|walks
```

Notice that the non-terminal `sentence` has a single option consisting of multiple non-terminals, whereas the others non-terminals each consist of multiple options.

Rules may be duplicated for the same non-terminal to make a particular expansion more likely than others. For example, suppose the above grammar were modified as follows:

```
sentence::=article object verb
article::=the|a
object::=cat|cat|dog
verb::=runs|walks
```

This grammar would produce the same sentences as the original grammar, but sentences containing "cat" would be twice as likely to occur as sentences containing "dog."

In addition, for this assessment, you may assume the following about all BNF rules:

- Each line will contain *exactly one occurrence* of `::=` which will be the separator between the name of a non-terminal and its options.
- A pipe (`|`) will separate each option for a non-terminal. If there is only one option for a particular non-terminal (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 ignored.
- Symbols are case-sensistive. (For example, `<S>` would not be considered the same symbol as `<s>`.)
- A *terminal* is any symbol that does not appear on the left-hand side of a rule.
- The text before the "`::=`" is not empty, does not contain a pipe (`|`) character, and does not contain any whitespace.
- The text after the "`::=`" will be nonempty.

Program Behavior

In this assessment you will write a class that accepts a list of rules for a grammar in Backus-Naur Form and allows the client to randomly generate elements of the grammar. You will use **recursion** to implement the core of your algorithm.

We have provided you with a client program, `GrammarMain.java`, that handles the file processing and user interaction. This program reads a BNF grammar input text file and passes its entire contents to you as a `List` of `Strings`. You will write a class `GrammarSolver` that generates random results based on the rules provided.

GrammarSolver

Your `GrammarSolver` class should have the following constructor:

`public` **`GrammarSolver`**`(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 empty or if there are two or more entries in the grammar for the same non-terminal.

Your `GrammarSolver` should also implement the following public methods:

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

This method should return `true` if the given symbol is a *non-terminal* in the grammar and `false` otherwise.

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

`public 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

---

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

`public String[]` **`generate`**`(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 you should throw an `IllegalArgumentException`.

When generating a non-terminal symbol in your grammar, each of the rules on the right-hand side of the grammar should be applied with equal probability.

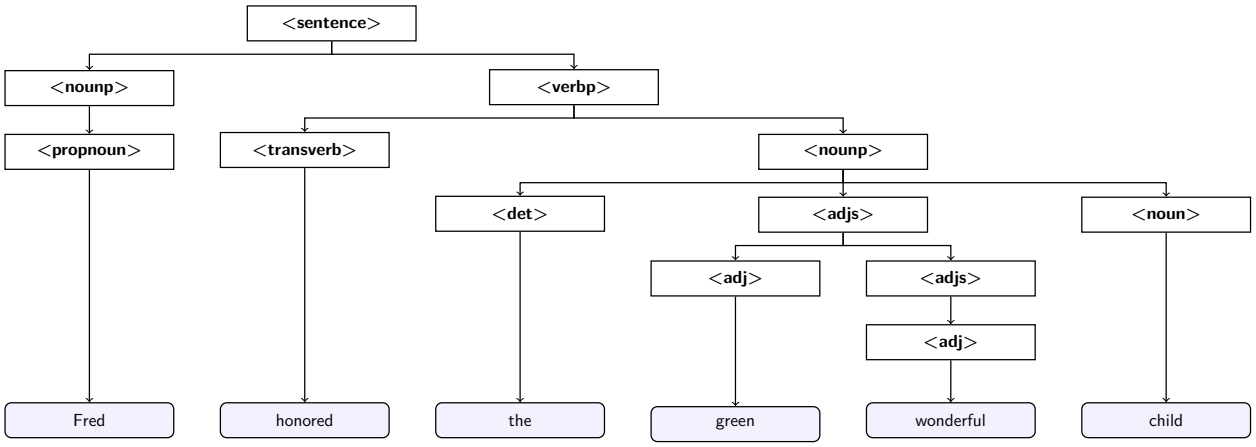
Each *written* rule should equally likely, but a rule may occur more often if it appears as an option more than once.

Sample Grammar and Executions

Complex BNF (sentence.txt)

```
<sentence>::=<nounp> <verbp>
<nounp>::=<det> <adjs> <noun>|<propnoun>
<propnoun>::=Hadi|Jazmin|Ali|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>::=taught|honored|waved to|helped
<intransverb>::=died|collapsed|laughed|wept
```

Example Random Sentence Diagram



Partial Example Execution (user input underlined)

```
Welcome to the cse143 random sentence generator.

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

Available symbols are:
[<adj>, <adjs>, <det>, <intransverb>, <noun>, <nounp>, <propnoun>, <sentence>, <transverb>, <verbp>]
What do you want generated (return to quit)? <sentence>
How many do you want me to generate? 5
Hadi found Jazmin
Spot helped the big cat
Elmo died
the green mother wept
the subliminal green man laughed

Available symbols are:
[<adj>, <adjs>, <det>, <intransverb>, <noun>, <nounp>, <propnoun>, <sentence>, <transverb>, <verbp>]
What do you want generated (return to quit)?
```

More example program executions are found at the end of the spec.

Implementation Guidelines

`GrammarSolver` Constructor

For this assessment, you **MUST** represent your grammar using a `Map`, where the *keys* of the map are the *non-terminals* of the grammar, and the *values* are the options for expansion the corresponding non-terminal. You should choose an appropriate data structure for the values in your `Map` to effectively represent the grammar rules and make the operations required by the class convenient and efficient.

`generate` Algorithm

The `generate` method will generate a random occurrence of a given non-terminal NT. You **MUST** use the following recursive algorithm in your implementation of this method:

> 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. If the symbol is a terminal, the expansion is simply the symbol itself. If the symbol is a non-terminal, you should generate an expansion using a recursive call.

Remember that it is acceptable to have a loop inside your recursion. (In fact, you will likely want one as part of this algorithm!) The directory crawler program from class will serve as a good guide for how to write this program. In that example, we iterated over the different files in a directory and used recursion to list the files in each subdirectory. For your `GrammarSolver`, you will iterate over the different symbols in the chosen role and use recursion to generate an expansion for each symbol. You may also find that you will want to use a public/private pair for this recursive task.

Testing Your Solution

We are providing another tool that is linked on the section for this assignment to check the output of your `generate` method to make sure it is producing valid output.

Remember to remove any debugging code when you submit.

You can test that the correct whitespace is produced from `generate` by using some non-whitespace character (e.g. ~) 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 the **`split` method of the `String` class**, which takes a `String` delimiter (e.g. "what to split by") as a parameter and returns your original large `String` as an array of smaller `String`s.

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".

You do not need to have a deep understanding of regular expressions to complete this assessment. Here are some specific regular expressions that will help you with particular splitting steps for your class:

- **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:

```
String line = "example::=foo bar |baz";
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:

```
String rules = "foo bar|baz   |quux   mumble";
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:

```
String rule = "the quick      brown          fox";
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:

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

Development Strategy and Hints

The `generate` method is the most difficult, so we strongly suggest you write it last. Remember that it is helpful to tackle difficult methods using "iterative development" where you solve simple versions of the problem first.

Random programs can be difficult to validate correctness, and the `generate` method you will implement uses randomness to decide which rule for a given non-terminal to use. To help you debug and validate your output, we have provided a grammar verifier tool on the course website that verifies your output follows the grammar rules (but ignores whitespace).

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

Code Quality Guidelines

In addition to producing the behavior described above, your code should be well-written and meet all expectations described in the grading guidelines, Code Quality Guide, and Commenting Guide. For this assessment, pay particular attention to the following elements:

SortedMap

Because we want you to guarantee the keys of your map are sorted, we will ask you to use the `SortedMap<K, V>` interface for this assignment instead of the `Map<K, V>` interface. The `SortedMap` interface is essentially the same as the `Map` interface, except it requires the keys be sorted. This means `TreeMap` is a valid `SortedMap` implementation, but `HashMap` is not. You can use all the same methods on a `SortedMap` as you could on a `Map`.

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.

Exceptions

The specified exceptions must be thrown correctly in the specified cases. Exceptions should be thrown as soon as possible, and no unnecessary work should be done when an exception is thrown. Exceptions should be documented in comments, including the type of exception thrown and under what conditions.

Commenting

Each method should have a header comment including all necessary information as described in the Commenting Guide. Comments should be written in your own words (i.e. not copied and pasted from this spec) and should not include implemenation details.

Running and Submitting

If you believe your behavior is correct, you can submit your work by clicking the "Mark" button in the Ed assessment. You will see the results of some automated tests along with tentative grades. **These grades are not final until you have received feedback from your TA.**

You may submit your work as often as you like until the deadline; we will always grade your most recent submission. Note the due date and time carefully—**work submitted after the due time will not be accepted**.

Getting Help

If you find you are struggling with this assessment, make use of all the course resources that are available to you, such as:

- Reviewing relevant examples from class
- Reading the textbook
- Visiting office hours
- Posting a question on the message board

Collaboration Policy

Remember that, while you are encouraged to use all resources at your disposal, including your classmates, **all work you submit must be entirely your own**. In particular, you should **NEVER** look at a solution to this assessment from another source (a classmate, a former student, an online repository, etc.). Please review the full policy in the syllabus for more details and ask the course staff if you are unclear on whether or not a resource is OK to use.

Reflection

In addition to your code, you must submit answers to short reflection questions. These questions will help you think about what you learned, what you struggled with, and how you can improve next time. The questions are given in the file `GrammarSolverReflection.txt` in the Ed assessment; type your responses directly into that file.

Sample Execution #1 (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>, <adjs>, <det>, <intransverb>, <noun>, <nounp>, <propnoun>, <sentence>, <transverb>, <verbp>]
What do you want generated (return to quit)? <det>
How many do you want me to generate? 5
a
the
the
a
the

Available symbols to generate are:
[<adj>, <adjs>, <det>, <intransverb>, <noun>, <nounp>, <propnoun>, <sentence>, <transverb>, <verbp>]
What do you want generated (return to quit)? <nounp>
How many do you want me to generate? 5
Elmo
a green big pretentious green pretentious subliminal university
the pretentious cat
Jazmin
the pretentious subliminal mother

Available symbols to generate are:
[<adj>, <adjs>, <det>, <intransverb>, <noun>, <nounp>, <propnoun>, <sentence>, <transverb>, <verbp>]
What do you want generated (return to quit)? <sentence>
How many do you want me to generate? 20
a faulty dog laughed
Ali helped a wonderful dog
Spot collapsed
the green father wept
Spot laughed
Elmo taught Ali
the subliminal green man honored Fred
a wonderful faulty big father laughed
the faulty faulty university taught the faulty dog
Elmo helped the green university
Hadi helped the pretentious man
the pretentious man died
Ali laughed
the pretentious subliminal child found Hadi
Elmo wept
a wonderful wonderful faulty child collapsed
Spot found the subliminal subliminal pretentious university
the green father helped the wonderful cat
a faulty television wept
the faulty mother laughed

Available symbols to generate are:
[<adj>, <adjs>, <det>, <intransverb>, <noun>, <nounp>, <propnoun>, <sentence>, <transverb>, <verbp>]
What do you want generated (return to quit)?
```

Sample Execution #2 (user input underlined)

```
Welcome to the cse143 random sentence generator.

What is the name of the grammar file? sentence2.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)?
```