Creative Project 2: Grammar Solver

Specification

Learning Objectives

- Implement recursive methods to solve a naturally recursive problem.
- Implement a public-private recursive pair
- Choose an appropriate data structure to represent specified data.
- Produce clear and effective documentation to improve comprehension and maintainability of a class
- Write a class that is readable and maintainable, and that conforms to the provided guidelines for style and implementation

Background

Formal Language

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 is considered a valid construction 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.

Grammar

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 English grammar, 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”
- “the dog runs”
- “the dog walks”
- “a cat runs”
- “a cat 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 defines a set of rules for a non-terminal that looks like the following:

```
non-terminal::=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 simple 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 other non-terminals each consist of multiple options.

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 have any special meaning. There will be at least one whitespace character between each part of a single rule (e.g., `Sentence::=Article Object Verb`). Extra whitespace should be ignored.
- Symbols are case-sensitive.
    - 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 project, you will write a class `GrammarSolver` 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.

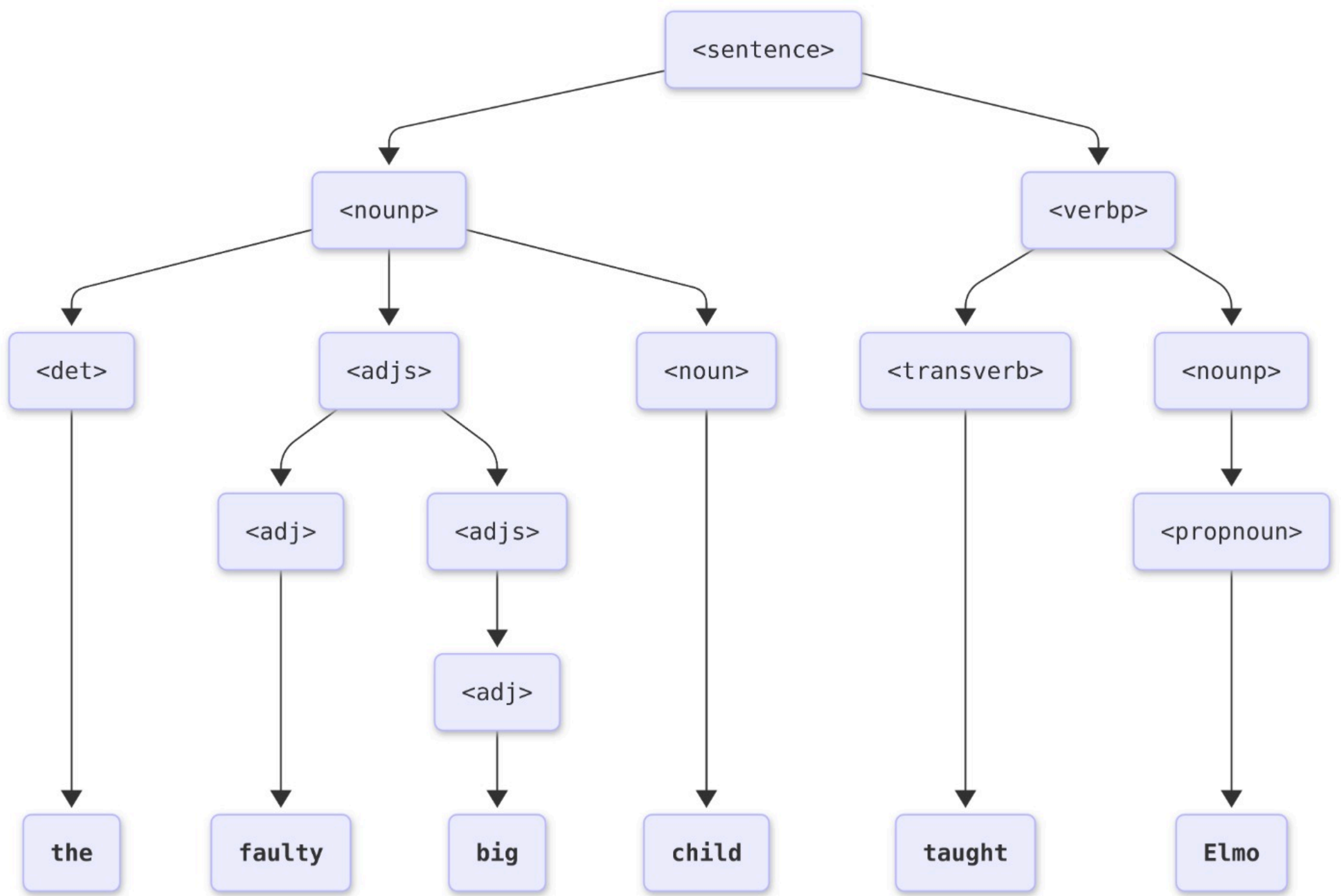
We have provided you with a client program, `GrammarMain.java`, that handles the file processing and user interaction. This program reads a BNF grammar input text file and passes its entire contents to you as a list of strings.

Client Program

Complex BNF (english.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



This diagram shows one possible sequence of rule choices used to generate the sentence “the faulty big child taught Elmo” from the starting non-terminal `<sentence>`. First, `<sentence>` expands into `<nounp> <verbp>`. The first `<nounp>` expands into `<det> <adjs> <noun>`. Then, `<det>` becomes the terminal `the`. The `<adjs>` symbol expands recursively into `<adj> <adjs>`. The first `<adj>` becomes the terminal `faulty`. The remaining `<adjs>` then expands into `<adj>`, and that `<adj>` becomes the terminal `big`. Next, `<noun>` becomes the terminal `child`. This completes the first noun phrase, "the faulty big child."

The `<verbp>` then expands into `<transverb> <nounp>`. The `<transverb>` becomes the terminal `taught`. The second `<nounp>` expands into `<propnoun>`, and `<propnoun>` becomes the terminal `Elmo`. Reading the terminal words from left to right gives the final generated sentence: “the faulty big child taught Elmo.”

Partial Example Execution (user input underlined and bolded)

```
Welcome to the CSE 123 random sentence generator!

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

Available symbols to generate are:
[<adj>, <adjs>, <det>, <intransverb>, <noun>, <nounp>, <propnoun>, <sentence>, <transverb>, <verbp>
What do you want to generate (Enter to quit)? <sentence>
Enter 1 for basic or 2 for complex: 1
```

```
How many do you want me to generate? 5
Jazmin wept
Ali honored Spot
the subliminal big man wept
a subliminal dog honored Jazmin
the faulty big child taught Elmo

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

Sample Execution #2 (user input underlined and bolded)

▶ Expand

Sample Execution #3 (user input underlined and bolded)

▶ Expand

Required Methods

Your `GrammarSolver` class must include the following:

```
public GrammarSolver(List<String> grammar)
```

- Initialize a new grammar solver over the given BNF grammar.
  - Each element of the list should be a definition in the following form (described in more detail above): `non-terminal::=rule|rule|rule|...|rule`
- This method should throw an `IllegalArgumentException` if any of the following cases are met:
  - `grammar` is null
  - `grammar` is empty
  - `grammar` defines the same non-terminal more than once.

```
public boolean contains(String symbol)
```

- Return `true` if the given `symbol` is a non-terminal in the grammar and `false` otherwise.
  - For example, when using the simple grammar described previously, return `true` for a call of `contains("Article")` and `false` for a call of `contains("<foo>")` or `contains("cat")` ("cat" is a terminal in the language).
- Throws an `IllegalArgumentException` if `symbol` is `null`.

```
public Set<String> getSymbols()
```

- Return all non-terminal symbols of your grammar as a sorted set of strings.
  - For example, when using the previous grammar, `getSymbols()` would return a set containing the four elements: `["Article", "Object", "Sentence", "Verb"]` in this order.

```
public String[] generateBasic(String symbol, int times)
```

- Randomly generates the specified number of strings for a provided grammar symbol, returning them as a `String[]`. For any given non-terminal symbol, each of the rules on the right-hand side of each line should be applied with **equal** probability.
  - 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.
  - **NOTE:** Each written rule should be equally likely to be chosen, but a rule may occur more often if it appears as an option more than once.
- This method should throw an `IllegalArgumentException` if any of the following cases are met:
  - `symbol` is null
  - `symbol` is not a non-terminal in the grammar
  - `times` is negative

```
public String[] generateComplex(String symbol, int times)
```

- Randomly generates a specified number of strings for a provided grammar symbol, returning them as a `String[]`. For any given non-terminal symbol, each of the rules on the right-hand side of the grammar should be applied with a complex probability depending on the creative extension you selected.
- This method should throw an `IllegalArgumentException` if any of the following cases are met:
  - `symbol` is null
  - `symbol` is not a non-terminal in the grammar
  - `times` is negative

Creative Extension

So far, our algorithm for generating symbols is rather boring: each rule is applied with uniform probability. For the creative extension, we will introduce some complexity in the generation! You should implement one of the three following options as your creative extension:

Shorter output

▶ Expand

Blacklisting rules

▶ Expand

Weights

▶ Expand

Recursion

To earn a grade higher than N on this assignment, **your algorithms must be implemented *recursively*.** You will want to utilize the *public-private pair* technique discussed in class. You are free to create any helper methods you like, but the core of your algorithm must be recursive.

Remember that when we say "your algorithms must be recursive," we do not mean "loops are forbidden". We simply mean that the main part of the algorithm must be recursive. You can still use loops where appropriate, as long as they contribute toward your main recursive algorithm.

Development Tips

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 is a string that uses 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. We have provided three class constants storing the specific regular expressions that will help you with particular splitting steps for your class:

- **Splitting Non-terminals from Rules in their Definitions.** Given `String definition`, to split `definition` based on where "::=" occurs, you could use the regular expression "::=" (since you are looking for these literal characters). We define this regular expression as the class constant `DEFINITION_SEPARATOR`. For an example of using it:

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

- **Splitting Different Rules**. Given `String rules`, to split `rules` based on where the '|' character is, your regular expression will look similar to the above, except that, in regular expressions, '|' is a special character. So, we need to 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 `String`s). We define this regular expression as the class constant `RULE_SEPARATOR`. For an example of using it:

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

- **Splitting Apart a Single Rule.** Given `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", a space, newlines, etc. And by adding "+" afterwards, the regular expression is interpreted as "one or more whitespace". We define this regular expression as the class constant `TOKEN_SEPARATOR`. For an example of using it:

```
String rule = "the quick      brown         fox";
String[] pieces = rule.split(TOKEN_SEPARATOR); // ["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 `String str`, we can create a new `String` that omits all leading and trailing whitespace using the method `trim()`:

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

Implementation Guidelines

For this assignment, you should follow the Code Quality guide when writing your code to ensure it is readable and maintainable. In particular, you should focus on the following requirements:

- Avoid making objects you don't need. Look for opportunities to construct objects and reuse them elsewhere. This can be tricky to spot, but in particular, you should review your code statements that instantiate `new` objects or call other methods that do so.
- You should make all of your fields private, and you should reduce the number of fields to only those that are necessary for solving the problem.
- Fields should always be initialized inside a constructor or method, never at declaration.
- Exceptions should be thrown as soon as possible, and no unnecessary work should be done when an exception is thrown.
- You should comment your code following the Commenting Guide. You should write comments with basic info (a header comment at the top of your file), a class comment for your

`GrammarSolver` class, and a comment for every method.
    - Make sure to avoid including *implementation details* in your comments. In particular, for your object class, a *client* should be able to understand how to use your object effectively by only reading your class and method comments, but your comments should maintain *abstraction* by avoiding implementation details.
    - Continuing with the previous point, keep in mind that the client should **not** be aware of what implementation strategy your class/methods utilize.
- All methods present in your class that are not listed in the specification must be private.

Spec Walkthrough

An error occurred.

Unable to execute JavaScript.

Grammar Solver

Reflection

For this week's reflection, we would like everyone to read this short passage and then watch and engage with the following video.

Natural Language Processing (NLP) is the computational study of human language, aiming to enable machines to interpret, generate, and interact using natural language. In the early days of the field, it actually began with hand-crafted linguistic rules like the BNFs you just worked with. However, scientists found that this approach had poor scalability and brittle handling of real-world ambiguity. Nowadays, Large Language Models (LLMs) dominate the field.

Watch Large Language Models explained briefly (7m 57s)

An error occurred.

Unable to execute JavaScript.

Question 1

Please respond to **one** of the following two prompts in its entirety:

Prompt 1: Provide at least one meaningful similarity and at least one meaningful difference between the BNF generation you completed in this assignment, and how modern LLMs work now. Make sure to draw connections between this assignment and the video above!

Prompt 2: What is one other creative extension you could implement in your GrammarSolver class to continue to improve the generated strings, outside of those listed? Please describe how your extension would affect the grammar structure and/or the algorithm for generating strings, and also

what additional information you would need to keep track of to implement this extension.

Please label which prompt you are answering. For full credit, all questions within a prompt must be clearly answered. In particular, we suggest using the ACE (answer, cite, and explain) format. For a meaningful response, it may be helpful to pose some counterexamples, connect in terms of your own experience, add additional support from course materials, and be as specific as possible in your own reasoning

No response

Question 2

REQUIRED
You MUST answer this question to receive credit for the assignment

Which extension did you implement?

- ○ Shorter output
- ○ Blacklisting rules
- ○ Weights

Question 3

The following questions will ask that you practice **metacognition** to reflect on the topics covered on this assignment and your experience completing it. For each question, focus on your plan and/or process for working through the assignment along with the CS concepts. Think about things like how you organized your working time, what sorts of things tended to go wrong, and how you dealt with those errors or mistakes.

Describe how you represented non-terminal symbols and rules in your program. Why did you choose to represent the symbols and rules in this way? What is one other way that you could have represented the symbols and rules, and what is one benefit and drawback to this alternate approach?

No response

Question 4

Do you think the algorithm you used to build grammars would have been easier, harder, or neither to implement using iteration instead of recursion? Why?

No response

Question 5

What skills did you learn and/or practice with working on this assignment?

No response

Question 6

What did you struggle with most on this assignment?

No response

Question 7

What questions do you still have about the concepts and skills you used in this assignment?

No response

Question 8

About how long (in hours) did you spend on this assignment? (Feel free to estimate, but try to be close.)

No response

Question 9

Was any part of the specification or requirements unclear? If so, which part(s), how was it unclear, and how could it have been made more clear?

No response

Question 10

[OPTIONAL] Do you have any other feedback, questions, or comments about this assignment?

(Note that we may not be able to respond to questions here, so please post on the message board if you would like a response!)

No response

Final Submission

Final Submission

Fill out the box below and click "Submit" in the upper-right corner of the window to submit your work.

Question

I attest that the work I am about to submit is my own and was completed according to the course Academic Honesty and Collaboration policy. If I collaborated with any other students or utilized any outside resources, they are allowed and have been properly cited. If I have any concerns about this policy, I will reach out to the course staff to discuss *before* submitting.

(Type "yes" as your response.)

No response