CS320 Programming Languages
Assignment 3
Due: March 16th, 2020 at 11:59 PM

Objectives:

- Apply the program paradigms to simple problems and Practice new imperative programming language syntax
- This program focuses on programming with recursion.
- Compare the use of imperative and declarative programming languages
- Reflect on the choice of a suitable programming paradigm and language for a given problem or domain

Part 1- [80 points]

Tasks:

Write a python program that reads an input file with a grammar in a BNF (description Below) and allow the user to randomly generate elements of the grammar. You will use **recursion** to implement the core of your algorithm.

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

User inputs are in bold

```
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)? <np>
How many do you want me to generate? 4

a wonderful father
the faulty man
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? 3

Sally laughed
Elmo honored a faulty television
Elmo honored Elmo
```

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.

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.

What is a BNF for a Grammar?

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).***

As we learned the grammar contains symbols know as *terminals* because they represent fundamental words of the language. **A terminal** in the English language might be the word "boy" or "run". 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 "Mario honored the coach" because the words "Mario" and "coach" 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".

What to Submit

Turn in files named grammergenerator.py

How it will be graded

To achieve a full grade for this program you need to

- Submit a functional syntax errors free program. The program should work with any file that contains BNF grammar. Two samples are available for you to test sentence.txt, simple.txt, and here is the sample runs for the expected output from each file(Expected 1 and Expected 2).
    - My testing will not be limited to these two files, I can use other files and your program should be working correctly.
- Utilizing recursion to implement your algorithm as described previously. Part of the grade will be based 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.
- You should follow good general style guidelines such as: decomposing code into appropriate functions; appropriately using control structures like loops and if/else; good variable names; and not having any lines of code longer than 80 characters.
- Comment your code descriptively in your own words at the top of your class, each function, and on complex sections of your code. Comments should explain each function's behavior, parameters, return and pre/post-conditions.

Useful Tips

- Any token that ever appears on the left side of the ::= of any line is considered a non-terminal,
- 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.
- **hint use a dictionary to store the contents of the** dictionaries 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.
- As you will need to deal with breaking strings into various parts. There are several ways to do this, one of them is to use the **split method**.
    - If the string you split begins with a space, you will get an empty string at the front of your list, so use the **strip 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 those yourself.

Part 2 [20 points]

- So far you have been introduced to different programming languages during your study in the program such as (Java, C++, Python, C, Assembly, Prolog and Scala soon). Some of these languages are Imperative programming languages and some others known as Declarative Programming languages.
  Each language is suitable paradigm for a given problem or domain based. For example, we have seen how simple to solve a logical puzzle like crossword using Prolog, if we think about the pseudocode to solve the same problem using any imperative paradigm we can recognize how simple is using prolog in such a domain.

Task

- Write one or two maximum paragraphs that present your personal reflection on the choice of a suitable programming paradigm and language for a given problem or a domain based on your experiences so far. In your reflection try to provide some examples to justify why you would choose one language over the other for this specific domain.

What to Submit

Please submit a myreflection.txt that has your one paragraph reflection.