COT 4120 Automata Theory/ Formal Languages

Final Project Report

Date: Thursday, November 18th, 2021

Project Topic: Implementation and visualization of procedure Context-Free Grammar (CFG) to Push-Down Automata (PDA)

Team Members: Oanh Le, Yen Le, Minh Tran

Description

Regarding its concept, the procedure Context-Free Grammar (CFG) is a type of formal grammar that generates all conceivable string patterns in a given formal language. Context-Free Grammar (CFG) is a context-free grammar defined by four tuples: G = (P, T, V, S). Meanwhile, Push-Down Automata (PDA) is a means of implementing a CFG in the same manner that DFA is used to create regular grammar. A DFA can only store a finite amount of data, but a PDA can store an endless amount of data. In this selected problem, we will visualize the procedure of CFG to PDA by C++. Every CFG may be translated to a PDA of the same value. Any string accepted by the CFG will be derived to the leftmost by the built PDA. Besides, there are two primary steps to convert CFG to PDA:

- **Step 1:** Convert the given productions of CFG into GNF.
- **Step 2:** Convert the productions of step 1 to PDA

Implementation

We used C++ as a programming language for this project. The data structure is a hash map that stores the states as key and respective productions as value. For example, the production S → abAB | B | aB will be stored as below in the map:

```
Key (char) | Value (set of string)
     S    |    [abAB, B, aB]
```

The user input should follow the constraints:

1. Use $ for null.
2. Each production rule is separated by a comma.
3. Use a dot to end the grammar.
4. No spaces in between the input.
5. The left-hand side contains only one uppercase letter.

Now we know the key data structure needed to implement the procedure from CFG to PDA, in the next step we would want to check if the CFG is already in the GNF form. If it is already in GNF form, we would convert it directly into PDA. Otherwise, we have to convert it to GNF first before producing its PDA form. In the below subsections, we describe our implementation to convert from CFG to GNF, and from GNF to PDA.

I. CFG Simplification

Regarding the conversion of the context-free grammar (CFG) to Greibach normal form (GNF), we first need to simplify the CFG by removing the null, unit, and useless productions. We initially implement a function to remove null or lambda represented as '$' in our codes. In the function `eliminateNull()`, we must first identify all nullable variables. A variable 'A' is said to be nullable if A→$. After we have identified all of the nullable variables, we can begin to build the non-nullable production. We add the original production as well as all the combinations of the production that may be produced by changing the nullable variables in the production by \$. If all of the variables on the RHS of the production are nullable, for example, A→\$, we do not include state A in the new grammar.

Secondly, we build the function `unitProductions()` to get rid of any unit productions existing in the grammar. This function is accountable for removing A->B by adding production A→a to the grammar rule whenever B->a occurs in the grammar, deleting A→B from the grammar, and repeating these two steps until all unit productions are removed through a for each loop.

Thirdly, we construct the function `uselessProductions()` to remove all the useless productions which never terminate. This function is responsible for identifying any nonterminal variables that would never lead to a terminal string, then excluding all of the productions in which the variable appears, and repeating until all useless productions are eliminated through a for each loop.

II. CFG to GNF

Next, we build an algorithm to turn the simplified CFG into GNF. Remember that GNF is in the form: $A \rightarrow aB_1B_2B_k... \ (k \geq 0)$

We first call the function `allNonTerminal()` which prompts all productions to have all nonterminal symbols or one terminal symbol. The algorithm accepts the following cases:

- ❖ Case 1: If the string on the RHS of the production has a length of 1, it means that this string is terminal because we already get rid of unit productions (for example, A→a). We will not change anything because it is in GNF form.
- ❖ Case 2: If the string on the RHS of the production has a length greater than 1, we will make everything become non-terminal. For example, S→aabB (while A→a and B→b) will become S→AABB. For now, we will not worry if the production is in the PDA form or not, as our next function call will solve this problem. In addition, if we find some terminals that are not a part of any productions, we will add a new one into our production set.

  For example, S→aAc,A→a will become S→AAC,A→a,C→c (C→c is a new production that is added to our original set)

Next, we will call the function `oneBeginTerminal()` that makes all the productions, which are not in the form of GNF, contain one terminal symbol at the beginning. For instance, we have S→BBBB, whereas B→b. In this case, we replace the first B and make S→bBBB which is eventually in GNF.

III. *From GNF to PDA:*

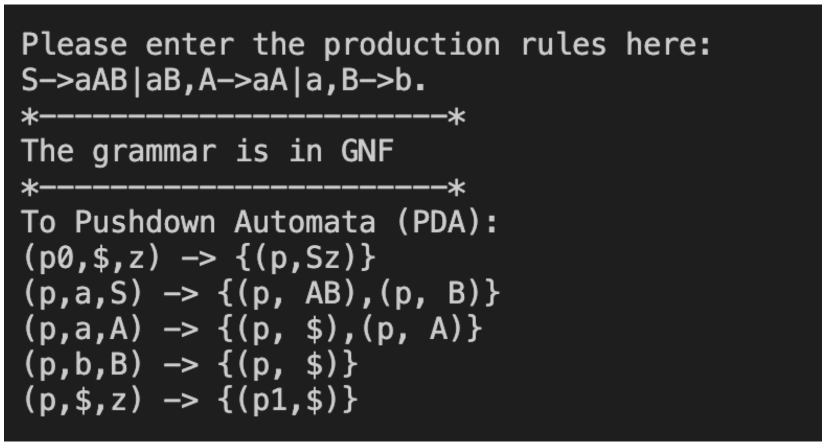
After successfully converting the grammar into Greibach Normal Form, we start to convert it into Pushdown Automata by calling the function `toPDA()`. To be more detailed, this function creates a hashmap that holds the strings on the right-hand side. The terminal will be stored as the key and the rest of the string will be stored as value. For example, aAAC | aB | a will be stored in the map as below:

```
Key (char) | Value (set of string)
    a    |    [AAC, B, $]
```

Each production in the language will have its own map to store the strings on the right-hand side. After everything is mapped, we will simply output the result by iterating through the map and printing it out in accordance with the PDA format.

Evaluation

Test case 1: Input is in GNF

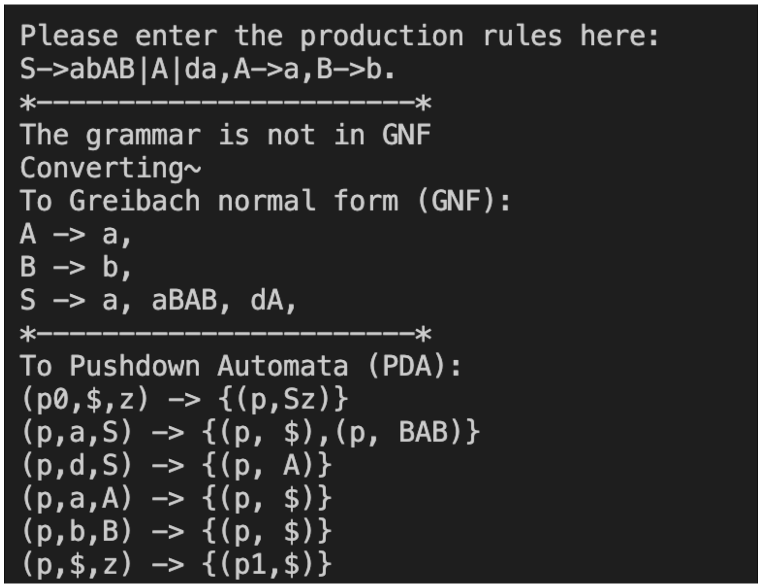


```
Please enter the production rules here:
S->aAB|aB,A->aA|a,B->b.
*----------------------*
The grammar is in GNF
*----------------------*
To Pushdown Automata (PDA):
(p0,$,z) -> {(p,Sz)}
(p,a,S) -> {(p, AB),(p, B)}
(p,a,A) -> {(p, $),(p, A)}
(p,b,B) -> {(p, $)}
(p,$,z) -> {(p1,$)}
```

Figure 1. GNF input

This output is right.

Test case 2: Input is not in GNF

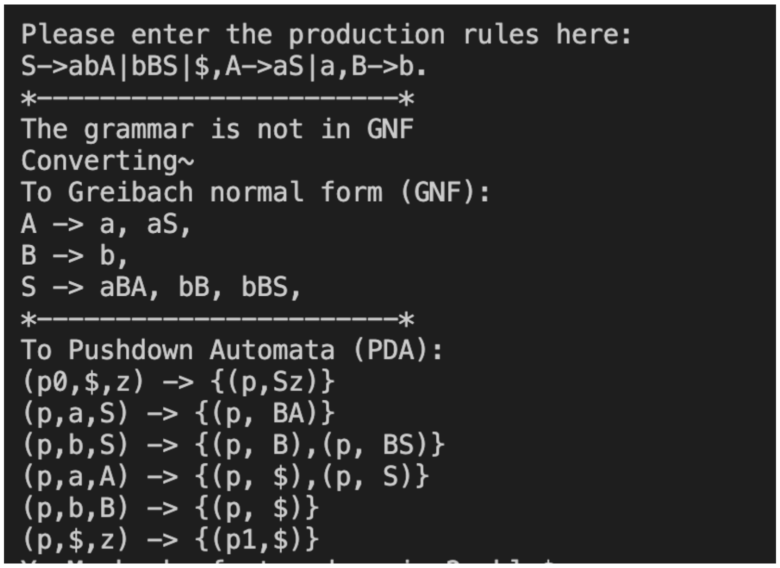


```
Please enter the production rules here:
S->abAB|A|da,A->a,B->b.
*----------------------*
The grammar is not in GNF
Converting~
To Greibach normal form (GNF):
A -> a,
B -> b,
S -> a, aBAB, dA,
*----------------------*
To Pushdown Automata (PDA):
(p0,$,z) -> {(p,Sz)}
(p,a,S) -> {(p, $),(p, BAB)}
(p,d,S) -> {(p, A)}
(p,a,A) -> {(p, $)}
(p,b,B) -> {(p, $)}
(p,$,z) -> {(p1,$)}
```

Figure 2. Non-GNF input

This output is right.

Test case 3: Input is not in GNF form with start symbol appears on the right hand side



```
Please enter the production rules here:
S->abA|bBS|$,A->aS|a,B->b.
*----------------------*
The grammar is not in GNF
Converting~
To Greibach normal form (GNF):
A -> a, aS,
B -> b,
S -> aBA, bB, bBS,
*----------------------*
To Pushdown Automata (PDA):
(p0,$,z) -> {(p,Sz)}
(p,a,S) -> {(p, BA)}
(p,b,S) -> {(p, B),(p, BS)}
(p,a,A) -> {(p, $),(p, S)}
(p,b,B) -> {(p, $)}
(p,$,z) -> {(p1,$)}
```

Figure 3. Non-GNF input with start symbol on the right hand side

This output is right.

Conclusion

Through this project, we managed to allow a user to input a grammar, verify whether the grammar is a context-free language, convert to Greibach Normal Form (GNF), and then convert to Pushdown Automata (PDA) by utilizing C++ to code our program. Unfortunately, our code production has some limited constraints on the user input. For example, it can only accept letter (a-z) as the terminal, while in practice we can enter 0 or 1. In addition, our code does not print out GNF in order. For instance, instead of printing out production of the start variable first, our code prints it out last (Figure 2). We are aware that it is because we store the variables in an ordered map, so everything is printed out in order (A-Z). If we are given more time to research, we believe that our code implementation can offer users more flexibilities in the input and a more readable output.