CS5252 Final Project

Yizhou Guo

Introduction

My project is to prove the equivalence of Context-Free Grammars (CFG) and Push-Down Automatas (PDA). To prove the equivalence, we can prove CFG$\in$PDA by converting every CFG to PDA, and prove PDA$\in$CFG by converting every PDA to CFG. During the weeks, I explored on the Internet on how CFG would convert to PDA, as well as how PDA would convert to CFG, and developed a program to convert any CFG to a easily-understandable PDA.

Convert PDA to CFG

In this part, I mainly gathered information from the internet. To convert PDA to CFG, the first step is to simplify the PDA. Particularly, we want to accept by empty stack and a single accepting state. We can easily do this by having a ϵ-transition between all accepting states to a virtual state. Also, we want to have the PDA that either push one symbol to the stack, or pop one symbol from the stack, but not both. This is also fairly straightforward. For example,

$$\delta(q_x, a, AB) = \delta(q_y, CD)$$

is equivalent to

$$\delta(q_x, a, A) = \delta(q_1, \epsilon)$$
$$\delta(q_1, \epsilon, B) = \delta(q_2, \epsilon)$$
$$\delta(q_2, \epsilon, \epsilon) = \delta(q_3, C)$$
$$\delta(q_3, \epsilon, \epsilon) = \delta(q_y, D)$$

Next, for all states q_i, q_j in the PDA (it is possible that $i = j$), introduce variable A_{ij}. For all A_{ii}, we have production rule $A_{ii} \to \epsilon$. For the rest, use the following rule:

$$A_{ij} \to A_{ik}A_{kj}$$

$A_{if} \to aA_{kp}b$ if $\delta(q_i, a, X) = \delta(q_k, \epsilon)$ and $\delta(q_i, b, \epsilon) = \delta(q_k, X)$, where X can be ϵ

The start state for the CFG will be A_{mn}, where q_m is the start state and q_n is the unique accepting state.

By using the rules above, we can convert any PDA to CFG.

Convert CFG to PDA

I spent most of my time on this part doing research, programming, and reasoning. A common method to convert CFG to PDA is:

For every production rule, for example, $A- > aBCd$, create a transition in PDA:

$$\delta(q_1, \epsilon, A) = \delta(q_1, aBCd).$$

For start symbol S, create a transition in PDA:

$$\delta(q_0, \epsilon, \epsilon) = \delta(q_1, S)$$

And for every variable A and character a, create transitions:

$$\delta(q_1, a, a) = \delta(q_1, \epsilon).$$

$$\delta(q_1, A, A) = \delta(q_1, \epsilon).$$

This is a simple and super nice proof, but personally, it is less useful. During week 6, we learned how to construct a PDA to accept certain Context-Free Languages, and the result is far from close to what we had constructed.

My idea is to write a program that convert CFG to PDA that for every transition, at most one symbol is pushed into or popped from the stack. To do this, I actually went to the wrong route at the first place. My first idea is, for every transition $A \to aBc$, construct

$$\delta(q_A, a, \epsilon) = \delta(q_A, A_a)$$

$$\delta(q_A, \epsilon, A_a) = \delta(q_A, BA_aA)$$

$$\delta(q_A, c, A_a) = \delta(q_A, \epsilon)$$

And for every state A, B, where A may equals to B, construct

$$\delta(q_A, \epsilon, B) = \delta(q_B, \epsilon)$$

It is quite obvious this is not going to work in a lot of cases. For example, in grammar G below:

$$S \to 0A$$

$$A \to 1$$

The produced PDA will be:

$$\delta(q_S, 0, \epsilon) = \delta(q_S, S_a) \tag{1}$$

$$\delta(q_S, \epsilon, S_a) = \delta(q_A, AS_aS) \tag{2}$$

$$\delta(q_A, 1, \epsilon) = \delta(q_A, \epsilon) \tag{3}$$

$$\delta(q_S, \epsilon, A) = \delta(q_A, \epsilon) \tag{4}$$

$$\delta(q_A, \epsilon, S) = \delta(q_S, \epsilon) \tag{5}$$

Consider string $011 \notin G$, by using rules 1,2,4,3,5,2,4,3,5, respectively, the string is recognized by the PDA.

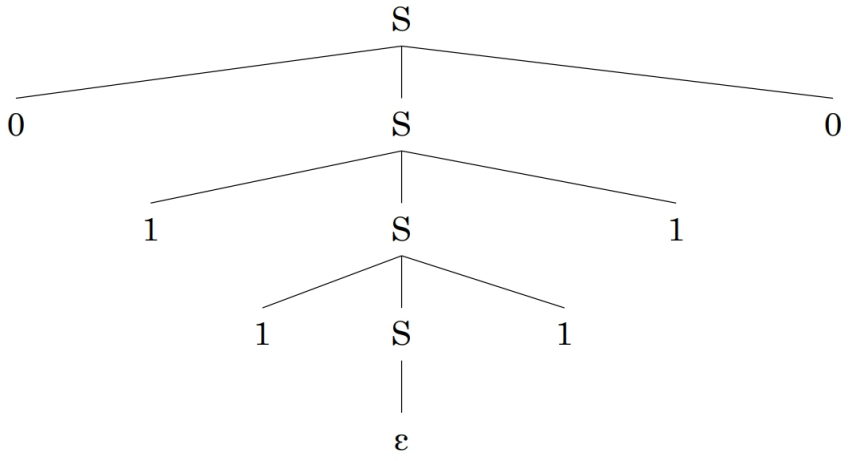
I came up with this because I was focused the example $\{w|w$ has two times as than b$\}$, and it would probably work on that example only. Later, when I figured out that this is wrong, I came up with the current approach that I am using:

At first, push a virtual symbol to the stack to avoid the stack being empty while we don't want it to. For each generation rule, assign a unique state for every character and variable in the rule. Recognize every character or variable will move the state forward, and we can only let it reach previous states by start recognizing a new generation rule. When a variable is reached, push a mark for current state, and transit to the corresponding state for the variable. When a generation rule is fully recognized, depend on what is top of the stack, go back to the intermediate state that 'called' the generation rule or go to the end state if we see the virtual symbol we pushed in at the start.

This is guaranteed to work since this use basically the same idea as parse

trees. Here is a example:

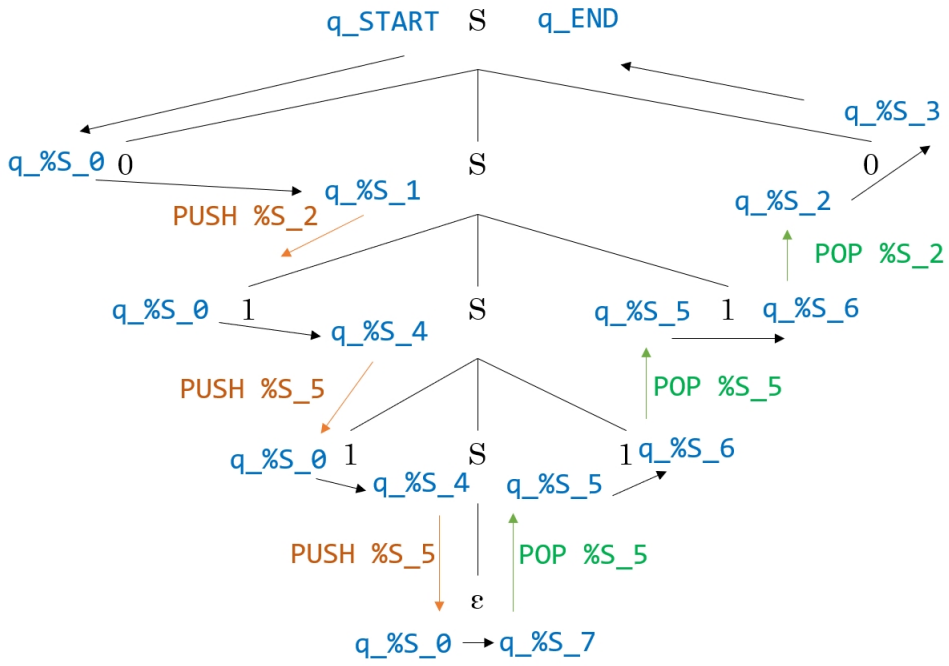
For CFL $\{ww^r|w \in (0,1)\}$, we have CFG $G : S \to 0S0|1S1|\epsilon$. Consider $011110 \in G$,we have parse tree as follows:



From the program we wrote, the following automata is generated (from an older version):

```
Start state: q_START
δ (q_START, ε , ε )= δ (q_%S_0, {ACCEPT})
δ (q_%S_0, 0, ε )= δ (q_%S_1, ε )
δ (q_%S_1, ε , ε )= δ (q_%S_0, %S_2)
δ (q_%S_2, 0, ε )= δ (q_%S_3, ε )
δ (q_%S_0, 1, ε )= δ (q_%S_4, ε )
δ (q_%S_4, ε , ε )= δ (q_%S_0, %S_5)
δ (q_%S_5, 1, ε )= δ (q_%S_6, ε )
δ (q_%S_0, ε , ε )= δ (q_%S_7, ε )
δ (q_%S_3, ε , %S_2)= δ (q_%S_2, ε )
δ (q_%S_3, ε , %S_5)= δ (q_%S_5, ε )
δ (q_%S_6, ε , %S_2)= δ (q_%S_2, ε )
δ (q_%S_6, ε , %S_5)= δ (q_%S_5, ε )
δ (q_%S_7, ε , %S_2)= δ (q_%S_2, ε )
δ (q_%S_7, ε , %S_5)= δ (q_%S_5, ε )
δ (q_%S_3, ε , {ACCEPT})= δ (q_END, ε )
δ (q_%S_6, ε , {ACCEPT})= δ (q_END, ε )
δ (q_%S_7, ε , {ACCEPT})= δ (q_END, ε )
Accept by empty stack only
```

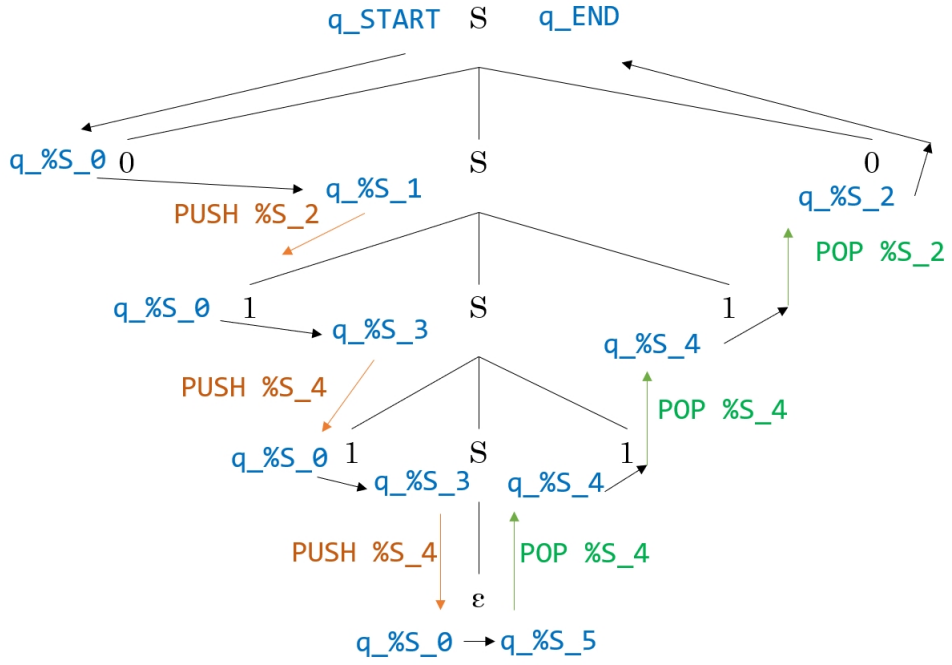
We can easily mark all our states and transitions corresponding to the parse tree:



However, there are some obvious redundant states in the parse tree, for example, it not necessary to have both q_{S_5} and q_{S_6}, since we can read a character and pop a stack symbol at the same time. Thus, I improved the program and generated the following automata:

```
Start state: q_START
δ (q_START, ε , ε )= δ (q_%S_0, {ACCEPT})
δ (q_%S_0, 0, ε )= δ (q_%S_1, ε )
δ (q_%S_1, ε , ε )= δ (q_%S_0, %S_2)
δ (q_%S_0, 1, ε )= δ (q_%S_3, ε )
δ (q_%S_3, ε , ε )= δ (q_%S_0, %S_4)
δ (q_%S_0, ε , ε )= δ (q_%S_5, ε )
δ (q_%S_2, 0, %S_2)= δ (q_%S_2, ε )
δ (q_%S_2, 0, %S_4)= δ (q_%S_4, ε )
δ (q_%S_4, 1, %S_2)= δ (q_%S_2, ε )
δ (q_%S_4, 1, %S_4)= δ (q_%S_4, ε )
δ (q_%S_5, ε , %S_2)= δ (q_%S_2, ε )
δ (q_%S_5, ε , %S_4)= δ (q_%S_4, ε )
δ (q_%S_2, 0, {ACCEPT})= δ (q_END, ε )
δ (q_%S_4, 1, {ACCEPT})= δ (q_END, ε )
δ (q_%S_5, ε , {ACCEPT})= δ (q_END, ε )
Accept by empty stack only
```

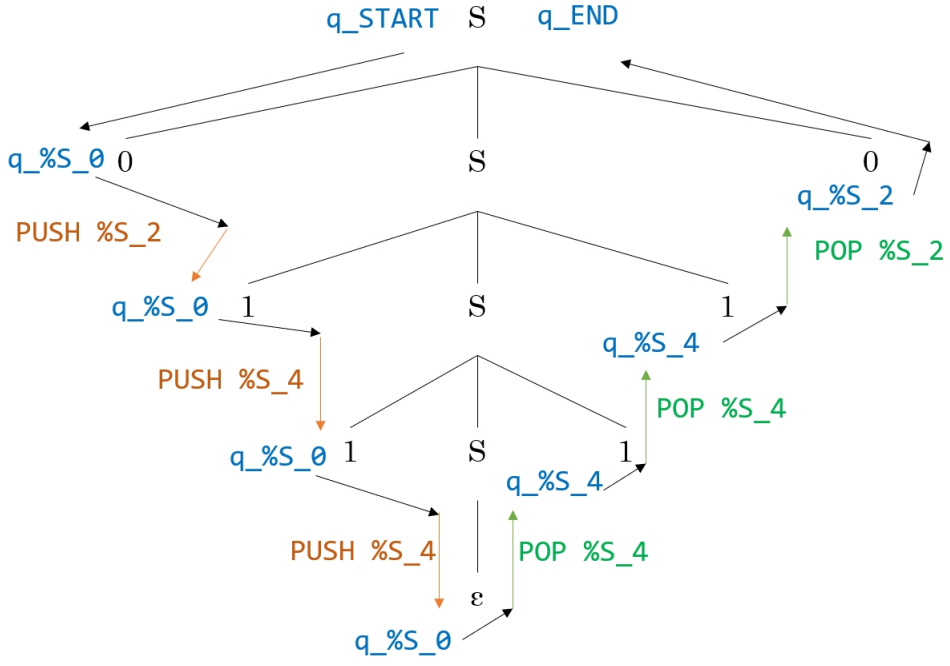
The corresponding states and transition in the parse tree for this updated automata will be like:



While this is improved, there is more space for improving that we can combine read a character and push a stack symbol in a single transition as well. Due to the limitation of time, I haven't implement this in the code. However, here is an ideal output from a further improved program:

```
Start state: q_START
δ (q_START, ε , ε )= δ (q_%S_0, {ACCEPT})
δ (q_%S_0, 0, ε )= δ (q_%S_0, %S_2)
δ (q_%S_0, 1, ε )= δ (q_%S_0, %S_4)
δ (q_%S_2, 0, %S_2)= δ (q_%S_2, ε )
δ (q_%S_2, 0, %S_4)= δ (q_%S_4, ε )
δ (q_%S_4, 1, %S_2)= δ (q_%S_2, ε )
δ (q_%S_4, 1, %S_4)= δ (q_%S_4, ε )
δ (q_%S_0, ε , %S_2)= δ (q_%S_2, ε )
δ (q_%S_0, ε , %S_4)= δ (q_%S_4, ε )
δ (q_%S_2, 0, {ACCEPT})= δ (q_END, ε )
δ (q_%S_4, 1, {ACCEPT})= δ (q_END, ε )
δ (q_%S_0, ε , {ACCEPT})= δ (q_END, ε )
Accept by empty stack only
```

The corresponding states and transition in the parse tree will be like:



The performance of the program for such CFG is actually fairly good. It has 15 transitions in improved version and 12 transitions in ideal version,

compared to 8 or 10-transition automatas written by myself and adapted from the lecture.

GitHub Repository (with code, instruction, and slides)

https://github.com/guoyizhou01/CS5252FinalProject

References

Chinese University of Hong Kong:
https://www.cse.cuhk.edu.hk/ siuon/csci3130-f18/slides/lec11.pdf
University of Notre Dame:
https://www3.nd.edu/ dchiang/teaching/theory/2016/notes/week06/week06b.pdf