

## Lecture 7: Pushdown Automata

*Lecturer: Abraham Ladha**Scribe(s): Rishabh Singhal*

## 1 Introduction

We mentioned previously how if we had a stack data structure, we could parse arithmetical expressions, like a classic data structures assignment. Lets do that. We are literally going to give an NFA a stack. We say a Pushdown Automata (PDA) is a tuple  $(Q, \Sigma, \Gamma, q_0, \delta, F)$

- $Q = \{q_0, \dots, q_k\}$  our set of states
- $\Sigma =$  input alphabet
- $\Gamma =$  Stack alphabet. By convention, we will usually set  $\Gamma = \Sigma \cup \{\$\}$ , where  $\$$  is a special symbol we denote as the stack canary.
- $q_0 \in Q$  is the start state
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) = \mathcal{P}(Q \times (\Gamma \cup \{\varepsilon\}))$  our transition function
- $F \subseteq Q$  is the set of final or accepting states.

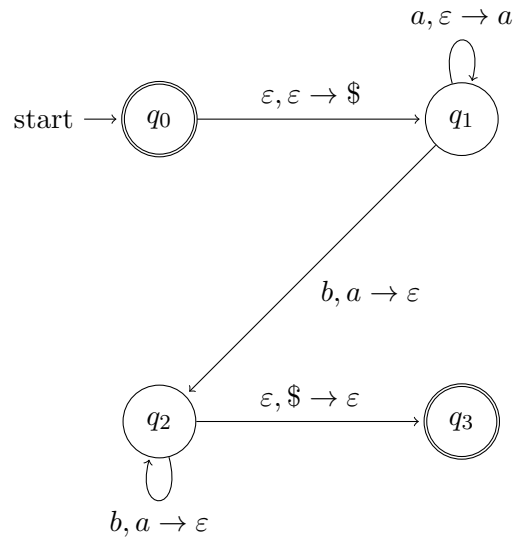
Notice that we have basically done as we said we would do. We just augmented an NFA to give it a stack. The transition function works as follows. You go from some state, optionally read a symbol off the input, and optionally pop the top of the stack. Then you transition to a new state and optionally push something onto the stack. When you see a transition of the form  $a, b \rightarrow c$ , you read  $a$  from input, pop  $b$  and push  $c$  onto the stack. Note that our defined PDA is explicitly nondeterministic. Deterministic PDAs exist, but unlike DFAs and NFAs, they are explicitly weaker. We will not cover them. Lets give some programming analogies to the types of transitions possible.

- $\varepsilon, \varepsilon \rightarrow \varepsilon$  We read nothing from the input, pop nothing, and push nothing. This means we really only change states, so this acts like an  $\varepsilon$ -transition in an NFA.
- $a, \varepsilon \rightarrow a; b, \varepsilon \rightarrow b$  We read  $a, b$  off the input, popped nothing, and pushed  $a, b$  respectively. We would only push exactly what we read, so this would push the input to the stack. If it was in a self-loop, it would push the entire input to the stack.
- $\varepsilon, a \rightarrow \varepsilon; \varepsilon, b \rightarrow \varepsilon$  Read nothing from the input, pop something, and push nothing. This would pop the top of the stack, whatever it may be. In a self-loop, it would dump the stack.
- $a, b \rightarrow b$  Read  $a$  off the input, pop  $b$  off the stack, then push  $b$  right back. This allows us to essentially transition off of "peeking" the top of the stack. We cannot read the top of the stack without popping it, but we can simulate this as popping it and then pushing it right back.

## 2 Examples

### 2.1 $\{a^n b^n \mid n \in \mathbb{N}\}$

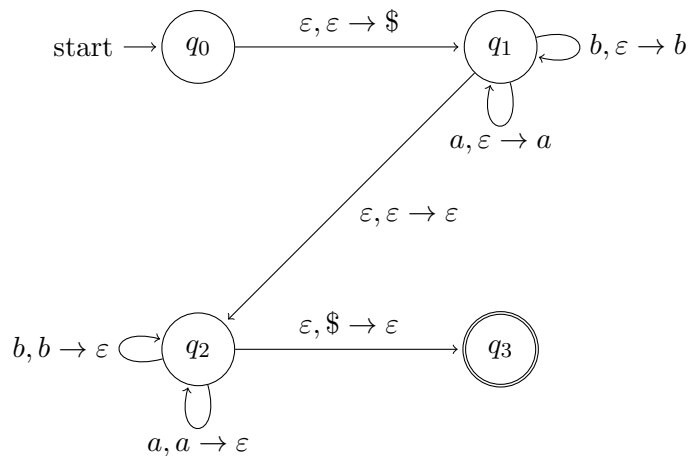
While reading  $a$ 's off the input, we are going to push them. If we read a  $b$ , we start reading  $b$ 's off the input, matching them to  $a$ 's popping off the stack, and we accept only if we read as many  $b$ 's as we previously pushed  $a$ 's. We don't have an inbuilt mechanism to determine if the stack is empty. That's why we begin almost every PDA by pushing  $\$$ . Then if we ever see our canary again, we know the stack is empty.



Note that we must also accept  $\epsilon$  so the start state is marked as accepting.

### 2.2 $\{ww^R \mid w \in \Sigma^*\}$

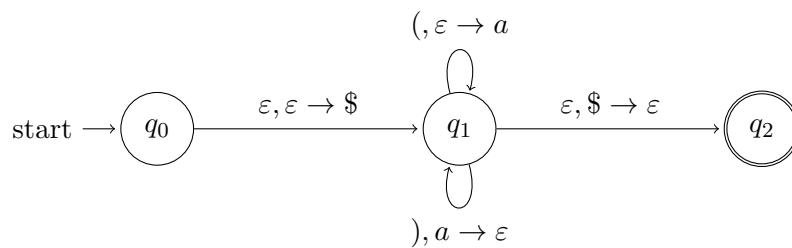
As you might suspect, our PDA will be similar, however before we could start matching the second half of our string by seeing a  $b$ . Here, we don't have that privilege. We can still solve it by nondeterministically guessing the midpoint of the input!



Note that all computation paths which incorrectly guess the midpoint of the string would implicitly reject. We also need not mark the start state as accepting as there exists a non-trivial computation path to accept  $\varepsilon$ . We would push the canary, epsilon transition, pop the canary, and accept.

### 2.3 Dyck Language

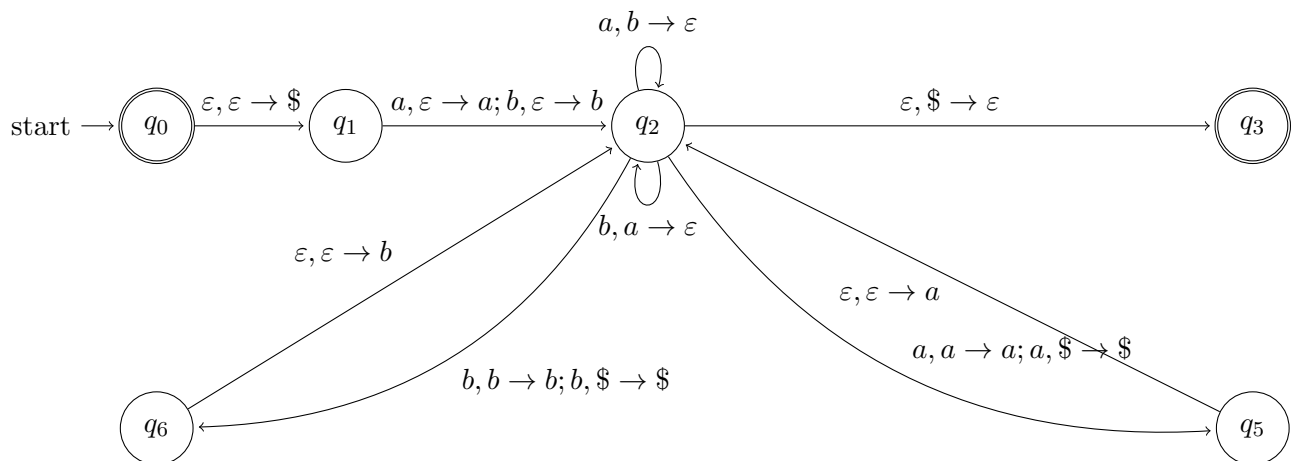
Recall that the Dyck language is the set of balanced parenthesis. You decide this language trivially with a counter. You increment your counter for every open you see, and decrement it for every close. If your counter ever goes negative, you reject. If your counter ends with anything other than zero, you reject. We will use the stack more simply here, as just a unary counter.



Here  $\Sigma = \{(\,, \,)\}$  and  $\Gamma = \{a, \$\}$ . Just for clarity, I chose  $\Gamma$  to be something else, as it doesn't particularly matter what we count with.

### 2.4 $\{w \in \Sigma^* \mid \#a(w) = \#b(w)\}$

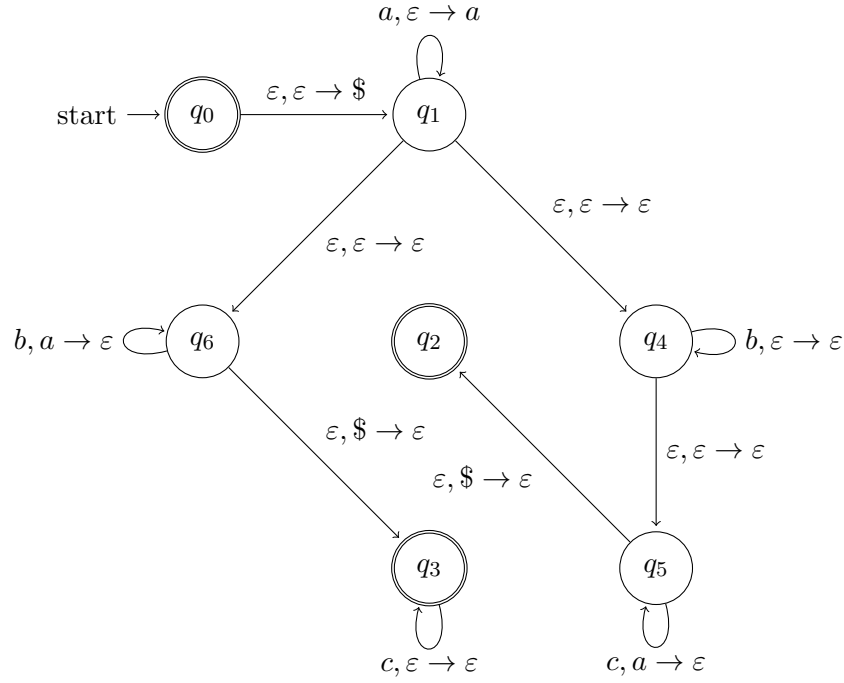
This will be similar to two PDAs we have seen, the Dyck language, and  $a^n b^n$ . However, instead of a positive counter, we would need a counter which could occasionally be negative. Different symbols on the stack will be used to represent the surplus in one direction or the other.



You should note we could do this PDA far simpler if we were allowed to push more than one symbol onto the stack per transition. We will show how to do this next lecture.

**2.5**  $\{a^i b^j c^k \mid i, j, k \in \mathbb{N}, i = j \text{ or } j = k\}$

Since the definition of the language contains an or, why don't we make two PDAs for each case, and then use nondeterminism to join them together.



Although we won't show it, this PDA requires nondeterminism. There is no DPDA to decide this language.

**3**  $\mathcal{L}(NFA) \subsetneq \mathcal{L}(PDA)$

We can convert any NFA into a PDA by simply preserving the topology of the states, and ignoring the stack. Some for some  $q_j \in \delta(q_i, a)$  in our NFA, we would have  $(q_j, \epsilon) \in \delta(q_i, a, \epsilon)$  in our PDA.



Our containment was strict because we gave PDAs for many languages we know not to be regular. In fact, all the examples we gave were languages we know to be context free. Next time we will show that  $\mathcal{L}(PDA) = \mathcal{L}(CFG)$ .