

Lecture 8: $\mathcal{L}(CFG) = \mathcal{L}(PDA)$

Lecturer: Abraham Ladha

Scribe(s): Samina Shiraj Mulani

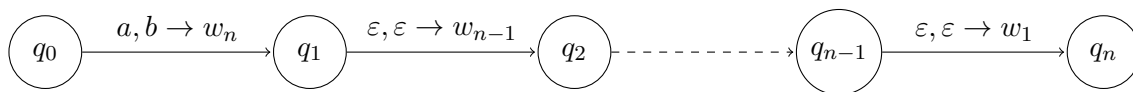
1 $\mathcal{L}(CFG) \subseteq \mathcal{L}(PDA)$

Recall the grammar with the following production rules $S \rightarrow aSb \mid \varepsilon$. which produces the language $\{a^n b^n \mid n \in \mathbb{N}\}$. As we produce strings, we get intermediate strings which are called “working strings” (example w.r.t aforementioned language - $aSb, aaaSbbb$). Once the working string does not have any non-terminals, that's the string produced by our choice of productions. We are trying to construct a PDA given a CFG. Consider some working string in the grammar. We will simulate part of it on the stack and part on the input. If we have a working string like $aaaSbbb$, anything that comes before the first non-terminal must be the prefix of the produced word. We don't need to keep this on the stack, but can just match it to the input.

Input \rightarrow $aaa:abbbb$ Stack \rightarrow $Sbbb\$$
 Working String \rightarrow $aaa:Sbbb$

- If top of the stack is a terminal, we pop it and advance input if it matches.
- If the top of the stack is a non-terminal, we pop it and non-deterministically choose its production and push it.

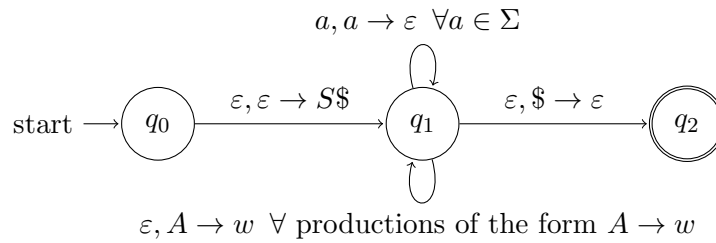
Observe that pushing a nonterminal's production on to the stack may involve pushing more than 1 symbol. We generalize and say that we can push $w_1 w_2 \dots w_n$ on to the stack by simulating the insertion in the PDA by adding more states as shown. $a, b \rightarrow w_1 \dots w_n$ would be



So, if we have $a, b \rightarrow abc$, top of the stack is now a . You should remember this.

1.1 CFG to PDA conversion

Our shortcut here allows us to represent the PDA for any CFG only using three states. If we didn't have our shortcut, each transition would need states for the length of the RHS of the production.



If I asked you to write a program to simulate a CFG, this might be non-trivial. Its interesting to note that actually the PDA is easier. Both PDAs and CFGs are nondeterministic, and we can have the nondeterminism of one simulate the nondeterminism of the other. The nondeterministic choice of productions becomes a nondeterministic choice of transitions.

1.2 Example

$$\{a^m b^n \mid m \geq n\}$$

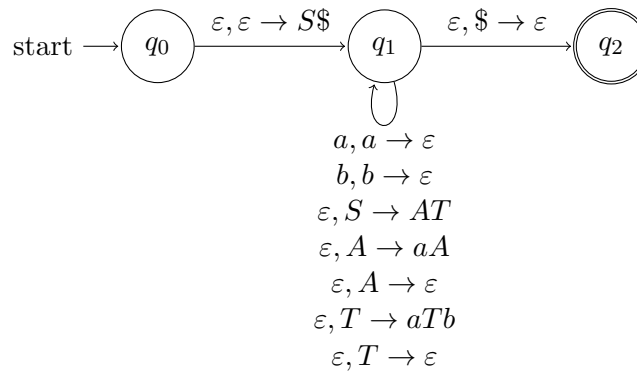
One correct CFG may be

$$S \rightarrow AT$$

$$A \rightarrow aA \mid \epsilon$$

$$T \rightarrow aTb \mid \epsilon$$

Here, A produces like a^* , and T produces like $a^n b^n$ matching. So $S \rightarrow AT$ will give us $a^m b^n$ with $m \geq n$.



Lets do a computation of aab to show the PDA accepts this grammar. Here, the underline of the input represents the symbol we are looking at, and the top of the stack is the leftmost.

| | | | | | | | | | | | |
|---------|-------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------|-------|
| Input → | aab | $\underline{a}ab$ | $\underline{aa}b$ | $\underline{a}ab$ | $\underline{aa}b$ | $\underline{a}ab$ | \underline{aab} | \underline{aab} | \underline{aab} | aab | aab |
| Stack → | $S\$$ | $AT\$$ | $aAT\$$ | $AT\$$ | $T\$$ | $aTb\$$ | $Tb\$$ | $b\$$ | $\$$ | empty stack | |

2 $\mathcal{L}(PDA) \subseteq \mathcal{L}(CFG)$

This proof is far harder than the previous one. Its easy to program a PDA to simulate a CFG. We will see its harder to “program” a CFG given a PDA. Instead of just doing an example, lets do a rigorous proof of correctness.

2.1 Make it nice

First, we have to make P nice, by modifying it in the following three ways.

1. P only has one accept state.
This trick also works for NFAs. Make a new final state, make all old accepting states as non-accepting, and then epsilon transition from all old accepting states to our single new one. This lets us assume there is only one accept state.
2. P accepts only with an empty stack.
Simply make sure we begin and end with pushing and popping $\$$. The old accept state should dump the stack before popping $\$$ to accept. This lets assume the computation ends how it begins, with an empty stack.
3. Each transition pushes or pops but not both.
Suppose we had a transition like $a, b \rightarrow c$. We can turn this into two transitions like $a, b \rightarrow \varepsilon$ and $\varepsilon, \varepsilon \rightarrow c$. This lets assume each move of the PDA either pushes or pops but not both.

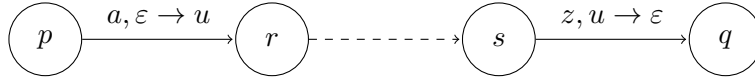
2.2 High level idea

For each $p, q \in Q$, make A_{pq} to represent strings which could take our PDA starting at state p and an empty stack, and ending on state q in an empty stack. The start variable should be A_{0f} where q_0 is the start state and q_f is the only final state. Since we are going from an empty stack to an empty stack, the first move must be a push and last move must be a pop. If the stack was never fully emptied from $p\dots q$ computation, then the first symbol pushed must be the last symbol popped, call it u . Let r, s be the next states making the computation path look like $pr\dots sq$. Here, r is the next state after p , and s is the state preceding q . Say a is what's first read of the input, and z is what's last read, resulting in $A_{pq} \rightarrow aA_{rs}z$.



If the stack was ever emptied from $p\dots q$ computation, we handle this as $\forall p, q, r \ A_{pq} \rightarrow A_{pr}A_{rq}$, where r would have been the state where the stack was empty. If it was empty at no other point, we can then inductively delegate A_{pr}, A_{rq} back to the first case. For PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, F = \{q_f\})$, we make the following CFG:

- $\forall p, q \in Q$ we add $A_{pq} \in V$ ($V =$ set of nonterminals).
- $S = A_{0f}$
- $\forall p \in Q, A_{pp} \rightarrow \varepsilon$
- $\forall p, q, r \in Q, A_{pq} \rightarrow A_{pr}A_{rq}$
- $\forall p, q, r, s \in Q, u \in \Gamma, a, z \in \Sigma \cup \{\varepsilon\}$, add rule $A_{pq} \rightarrow aA_{rs}z$ if there exist transitions in δ as defined previously like



Given a PDA P , we have constructed a grammar G . To prove the correctness of this construction, we need to prove:

$$A_{pq} \xRightarrow{*} x \iff x \text{ brings } P \text{ from } p \text{ (empty stack) to } q \text{ (empty stack).}$$

2.3 (\implies) direction

Given that $A_{pq} \xRightarrow{*} x$, we want to show that x brings P from p (empty stack) to q (empty stack). We proceed by proof by induction on the length of the derivation.

Base case: Derivations of length one

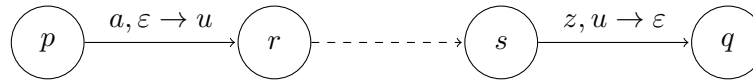
Out of all the rules, only one has no nonterminal on the right. There is only time for 1 production and that production must produce a string. $A_{pp} \rightarrow \varepsilon$. It is true that ε trivially brings P from p (empty stack) to p (empty stack).

Induction Hypothesis: Assume if $A_{pq} \xRightarrow{*} x$ in $\leq k$ derivations, then x brings P from p (empty stack) to q (empty stack).

We have to show: True for derivations of length $k + 1$

Assume $A_{pq} \xRightarrow{*} x$ in $k + 1$ steps. We have two cases.

1. First step in our derivation is of the form $A_{pq} \rightarrow aA_{rs}z$. By induction hypothesis, for $x = ayz$, $A_{rs} \xRightarrow{*} y$ in $\leq k$ steps, so y must bring P from r (empty stack) to s (empty stack). By construction we only have the rule $A_{pq} \rightarrow aA_{rs}z$ if:



So ayz brings P from p (empty stack) to r to s to q (empty stack). If y brings P from r (empty stack) to s (empty stack), then certainly it can bring P from r (just u in stack) to s (just u in stack). This means that $A_{pq} \xRightarrow{*} x$ and $ayz = x$ brings P from p (empty stack) to q (empty stack).

2. The first step in our derivation is of the form $A_{pq} \rightarrow A_{pr}A_{rq}$. By induction hypothesis, $x = vw$, $A_{pr} \xRightarrow{*} v$, $A_{rq} \xRightarrow{*} w$ in at most k steps. So v brings P from p (empty stack) to r (empty stack) and w brings P from r (empty stack) to q (empty stack). So, clearly $vw = x$ brings P from p (empty stack) to r (empty stack) to q (empty stack).

2.4 (\impliedby) direction

We want to show that if x brings P from p (empty stack) to q (empty stack), then $A_{pq} \xRightarrow{*} x$. We proceed by proof by induction on length of computation.

Base case: Computation of zero steps.

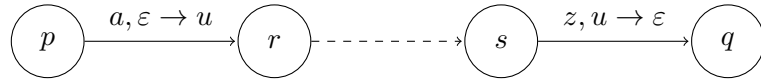
A computation of zero steps means we don't have time to switch states, so consider any such x with $A_{pp} \xRightarrow{*} x$. Zero steps also means that there is no time to read any input. So $x = \varepsilon$. But, we have the rule $A_{pp} \rightarrow \varepsilon$ as desired.

Induction Hypothesis: Assume its true for computation of length $\leq k$, that if x brings P from p (empty stack) to q (empty stack) in $\leq k$ computation steps then $A_{pq} \xRightarrow{*} x$

We have to show: True for computations of length $k + 1$.

Let x bring P from p (empty stack) to q (empty stack) in $k + 1$ steps. We have two cases.

1. Suppose the stack is only empty at the beginning and end. So the first symbol pushed must be the last symbol popped. By our construction for $x = ayz$



By induction hypothesis since y brings P from r (empty stack) to s (empty stack), then $A_{rs} \xRightarrow{*} y$ and we have the rule $A_{pq} \rightarrow aA_{rs}z$. So, $A_{pq} \xRightarrow{*} x$.

2. During the computation of length $k + 1$, the stack is empty at some middle point, lets say, at state r . The computations from p to r and r to q take at most k steps. By induction hypothesis, for $x = vw$, $A_{pr} \xRightarrow{*} v$ and $A_{rq} \xRightarrow{*} w$. Since we have the rule $A_{pq} \rightarrow A_{pr}A_{rq}$, $A_{pq} \xRightarrow{*} vw = x$.

3 Remarks

Recall that we say a language is context-free if there exists a CFG to produce it. We may now also say that a language is context-free if there exists a PDA to decide it. The PDAs decide exactly the same languages that CFGs produce. We did not give a formal proof that regular grammars produce only regular languages, but regular grammars are a strict superset of those which are context-free. This proof is an alternate way to see that the regular languages are also produced by context-free grammars. Every regular language can also be decided by a PDA, which is equivalent to some CFG.

This proof had many parts. We have to individually prove $\mathcal{L}(PDA) \subseteq \mathcal{L}(CFG)$ and $\mathcal{L}(CFG) \subseteq \mathcal{L}(PDA)$. We really only did half. For $\mathcal{L}(PDA) \subseteq \mathcal{L}(CFG)$, we gave a construction of a grammar G from a PDA P , and then we had to show $L(G) \subseteq L(P)$ and $L(P) \subseteq L(G)$. Then each of those had their own cases.