CS 4510 Automata and Complexity Feburary 7th 2024

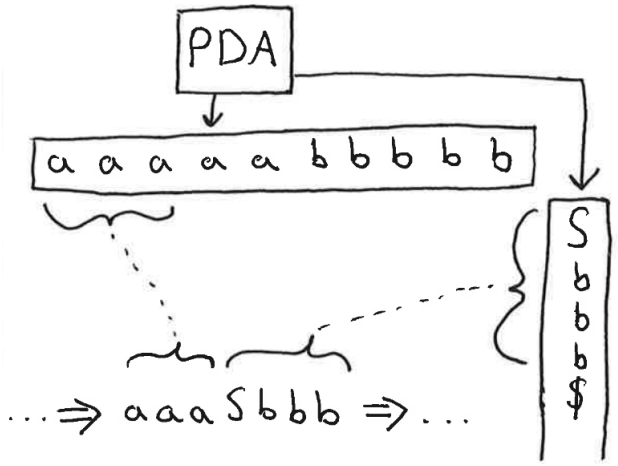
Lecture 8: $\mathscr{L}(CFG) = \mathscr{L}(PDA)$

Lecturer: Abrahim Ladha *Scribe(s): Samina Shiraj Mulani*

Context-free grammars and pushdown automata are very different devices, so it may come as a suprise to you that the languages producible by context-free grammars are exactly those which can be decided by pushdown automata. We prove their equivalence via a double set containment. Given a grammar, we produce an equivalent automata, and given an automata, we produce an equivalent grammar.

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

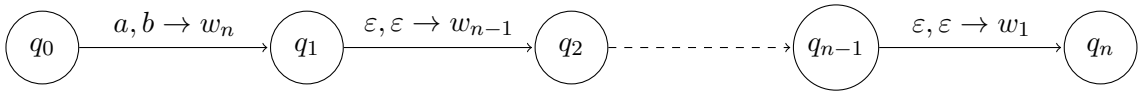
Recall the grammar with the following production rules $S \to aSb \mid \varepsilon$. which produces the language $\{a^n b^n \mid n \in \mathbb{N}\}$. As we produce strings, we get an intermediate sequence of strings which are called "working strings" (for example with respect to the aforementioned language - $aSb, aaaSbbb$). Once the working string does not have any remaining nonterminals, that is the string produced by that choice of productions. We are trying to construct a PDA given a CFG. We will have the sequential configurations of the PDA somehow encode the sequential working strings and sucessive computation steps simulate the application of sucessive productions. 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. This working string must produce a word that begins with aaa. We don't need to keep this on the stack, but can just match it to the input.



Here is the main simulation idea. The stack will contain part of a working string. If the top is a terminal, we pop it and just confirm it matches the input. If the top of the stack is a non-terminal, we pop it and nondeterministically choose a production and push it. Here, the nondeterminism of the grammar is simulated by the nondeterminism of the automata.

As long as there exists a sequence of productions to produce a word, there will exist a sequence of choices the machine can make, and atleast one correct computation branch so the machine will accept the word.

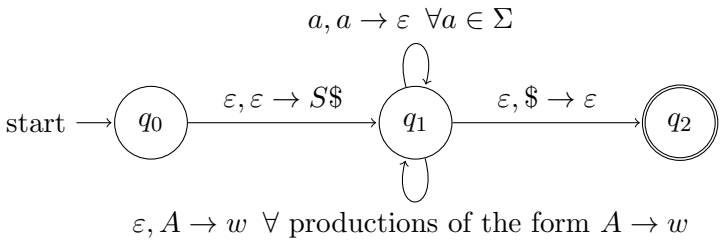
Observe that pushing a nonterminal's production on to the stack may involve pushing more than 1 symbol. We generalize and say that we can push $w_1w_2...w_n$ on to the stack by simulating the insertion in the PDA by adding more states as shown. $a, b \to w_1...w_n$ would be



So, if we have $a, b \to def$, top of the stack is now d. You should remember this convention.

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 production would need some number of states proportional to the length of their right-hand-sides.

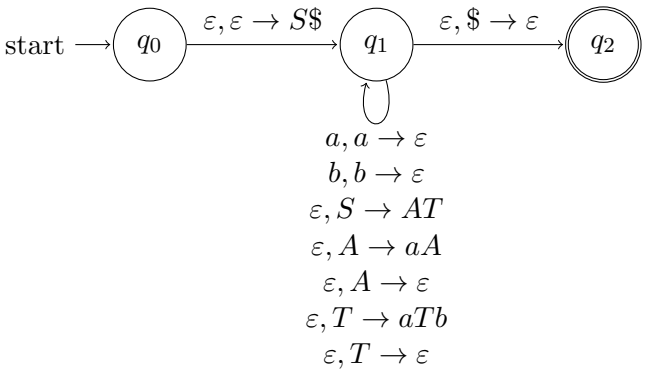


If I asked you to write a program to simulate a CFG, this might be non-trivial. Its interesting to note that actually the PDA is easier. Both PDAs and CFGs are nondeterministic, and we can have the nondeterminism of one simulate the nondeterminism of the other. The nondeterministic choice of productions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

1.2 Example

$\{a^m b^n \mid m \geq n\}$
One correct CFG may be
$S \to AT$
$A \to aA \mid \varepsilon$
$T \to aTb \mid \varepsilon$

Here, A produces like a^*, and T produces like $a^n b^n$ matching. So $S \to AT$ 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	$\underline{a}ab$	$\underline{a}a\underline{b}$	$a\underline{a}b$	$a\underline{a}\underline{b}$	$a\underline{a}b$	$aa\underline{b}$	$aa\underline{b}$	aab:	aab
Stack →	$S\$$	$AT\$$	$aAT\$$	$AT\$$	$T\$$	$aTb\$$	$Tb\$$	$b\$$	$\$$	empty stack

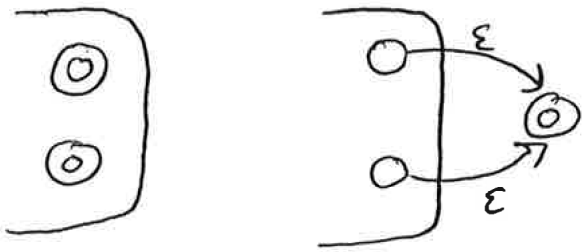
2 $\mathscr{L}(PDA) \subseteq \mathscr{L}(CFG)$

This proof more involved than the previous one. Its easy to program a PDA to simulate a CFG. We will see its harder to "program" a CFG to simulate a PDA. Instead of just doing an example, we wil 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

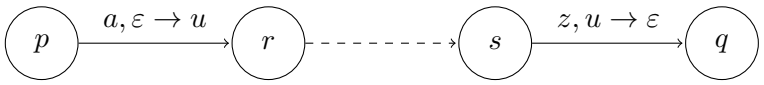
should dump the stack before popping \$ to accept. This lets assume the computation ends how it begins, with an empty stack.

3. Each transition pushes or pops but not both.
   Suppose we had a transition like $a, b \to c$. We can turn this into two transitions like $a, b \to \varepsilon$ and $\varepsilon, \varepsilon \to c$. 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 nonterminal A_{pq} to represent strings which could take our PDA starting at state p with an empty stack, and ending on state q with 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. We have two cases

- If the stack was never fully emptied during the computation from state $p...q$, then the first symbol pushed must also be the last symbol popped, call it u. Let r, s be the next states making the computation path look like $pr...sq$. Here, r is the next state after p, and s is the last state preceding q. Say a is whats first read of the input, and z is whats last read off the input. Our PDA structure may look like this.



  By assumption of our case, the computation from $r..s$ contains a u in the stack which is never emptied. If there is a computation which can take $r..s$ with just u in the stack, untouched, then there is a computation with can take $r..s$ empty stack to empty stack. We see then we may apply the production in our grammar to simulate this as $A_{pq} \to aA_{rs}z$.

- Suppose the stack was ever emptied from the $p...q$ computation. We add productions of the form $\forall p, q, r \quad A_{pq} \to A_{pr}A_{rq}$, 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. If there was another point the stack was emptied during the $p..r$ computation, then it recursively applies back to this case.

The following picture from Sipser may help understand what the two cases are.

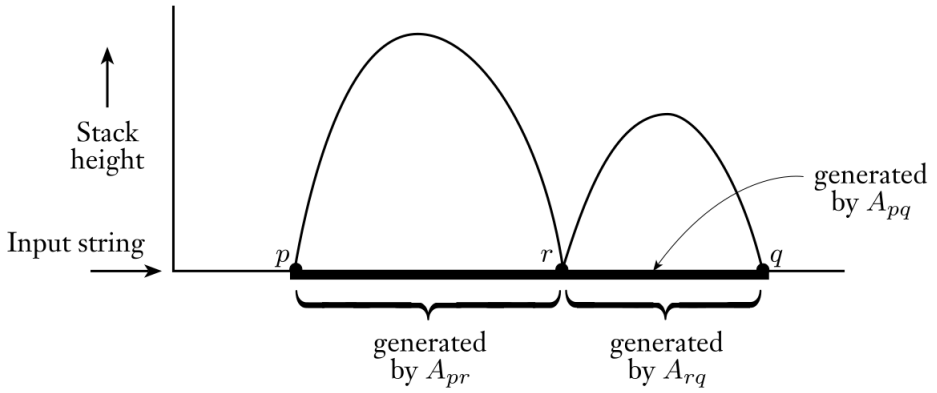
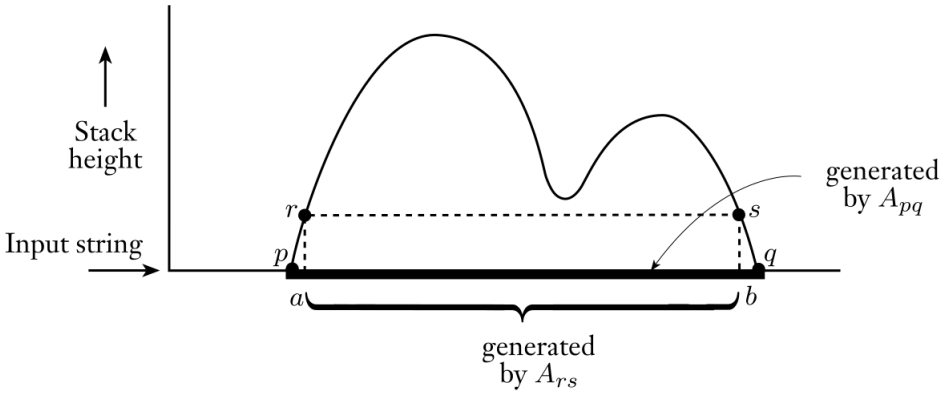
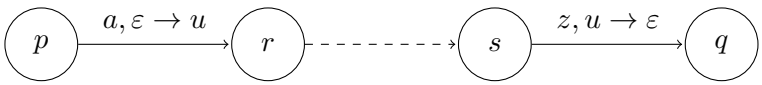


FIGURE 2.28
PDA computation corresponding to the rule $A_{pq} \to A_{pr}A_{rq}$



We now define the construction of an equivalent CFG G given PDA P. 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).
- $S = A_{0f}$
- $\forall p \in Q, A_{pp} \to \varepsilon$
- $\forall p, q, r \in Q, A_{pq} \to A_{pr}A_{rq}$
- $\forall p, q, r, s \in Q, u \in \Gamma, a, z \in \Sigma \cup \{\varepsilon\}$, add rule $A_{pq} \to aA_{rs}z$ if there exist transitions in δ as defined previously like



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

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

2.3 ($\Longrightarrow$) direction

Assume that $A_{pq} \stackrel{*}{\Rightarrow} x$. 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.

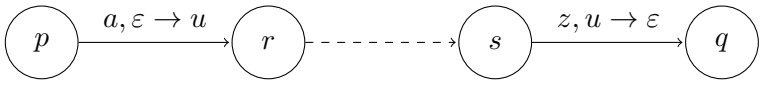
Base case: Derivations of length one

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

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

Assume $A_{pq} \stackrel{*}{\Rightarrow} x$ in $k+1$ steps. We want to show that x brings PDA P from p (empty stack) to q (empty stack). We have two cases.

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

2. The first step in our derivation is of the form $A_{pq} \to A_{pr}A_{rq}$. By induction hypothesis, $x = vw$, $A_{pr} \stackrel{*}{\Rightarrow} v$, $A_{rq} \stackrel{*}{\Rightarrow} w$ 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 ($\Longleftarrow$) direction

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

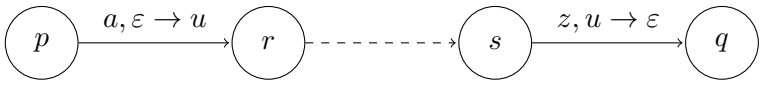
Base case: Computation of zero steps.

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

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

Let x bring P from p (empty stack) to q (empty stack) in $k+1$ steps. We want to show that $A_{pq} \stackrel{*}{\Rightarrow} x$

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 $A_{rs} \stackrel{*}{\Rightarrow} y$ and we have the rule $A_{pq} \to aA_{rs}z$. So, $A_{pq} \stackrel{*}{\Rightarrow} x$.

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

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 indidividually prove $\mathscr{L}(PDA) \subseteq \mathscr{L}(CFG)$ and $\mathscr{L}(CFG) \subseteq \mathscr{L}(PDA)$. We really only did half. For $\mathscr{L}(PDA) \subseteq \mathscr{L}(CFG)$, 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.