CSCI 341–Fall 2024: Lecture Notes
Set 10: Pushdown Automata

Edward Talmage

October 16, 2024

1 Definition

We would like machines to recognize Context-Free Languages in addition to grammars which generate them. The primary reason we saw for languages to not be regular was a lack of memory to count features of strings, so let us add some memory to a machine we already know.

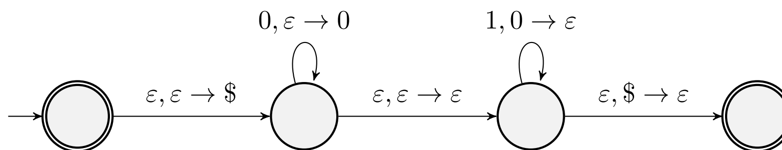
Definition 1. A Pushdown Automaton (PDA) M is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ with

- Q a finite set of states
- Σ an input alphabet
- Γ a stack alphabet
- $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$ a transition function
  - The output of any transition must be a finite set
  - We can push a finite sequence of characters to the stack by introducing a finite chain of states which pushes them one at a time, so we will typically act as if the codomain of δ is $\mathcal{P}(Q \times \Gamma^*)$.
- $q_0 \in Q$ a start state
- $F \subseteq Q$ a set of accept states

The idea is that a PDA is an NFA plus a LIFO stack for memory. Each transition can (optionally) pop a character from the stack and (optionally) push a finite sequence of characters onto the stack. We draw transitions with a label of the form $a, x \to \alpha$, where $a \in \Sigma_\varepsilon$ is a character from the input, $x \in \Gamma_\varepsilon$ is a character popped from the stack or ε indicating no pop, and $\alpha \in \Gamma^*$ is a string of characters to push onto the stack.

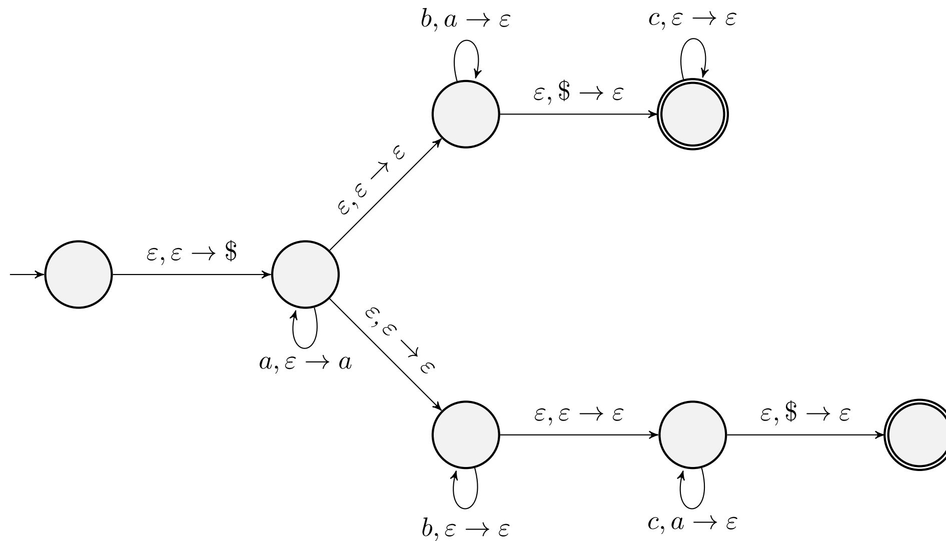
We say that PDA M *accepts* string w if there is a path from q_0 which consumes w and ends in an accept state. The language of M is the set of strings M accepts.

Examples: Consider the language $\{0^n 1^n \mid n \geq 0\}$.



The idea is that we push the leading 0's onto the stack, then match them up with 1's from the second part of the string. If there are ever any out of order characters or we run out of 0's before processing all the 1's, we will have no valid transition and reject. We use the special symbol \$, which we assume is not in the input alphabet, in our stack alphabet. We use this to mark the bottom of the stack, as we otherwise cannot tell at the end of the input whether the stack is empty, or we are just not popping anything.

Consider next the language $\{a^i b^j c^k \mid i = j \text{ OR } i = k, i, j, k \geq 0\}$. We will use a similar technique of marking the bottom of the stack, then non-deterministically split into two branches, one for each case.



Exercise: Give a PDA for each of the following languages:

(a) $\{w \mid w = w^R\}$

(b) $\{0^m 1^{m+n} 0^n \mid m, n \geq 1\}$

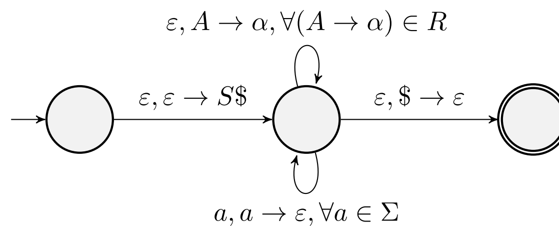
2 CFG-PDA Equivalence

The purpose of PDAs is, similar to the different structures we had for regular languages, to give us different ways to express context-free languges. Thus, we need to prove that every context-free language is recognized by some pushdown automaton, and that pushdown automata can only recognize context-free languages.

Theorem 1. *Given a CFG $G = (V, \Sigma, R, S)$, we can design a PDA M s.t. $L(M) = L(G)$.*

Idea:

- Generate a string on the stack, using the CFG rules, by starting with the start variable.
  - Non-determinism in the PDA explores all possible derivations in the CFG, so there is a PDA branch for every string in $L(G)$.
- Compare the stack string to the input string, accept if match.
  - The sneaky part here is that to expand all the variables on the stack, we must pop any terminals that are above variables. Popping those terminals loses them, which means we cannot later compare them to the input.
  - We can, however, go ahead and compare any terminals we find on the stack to the beginning of our input. Since there are no variables before those terminals, they will appear at the beginning of the generated string, which is the position in the input against which we compare.
- Now, PDA accepts (on some branch) iff CFG generates that input string.

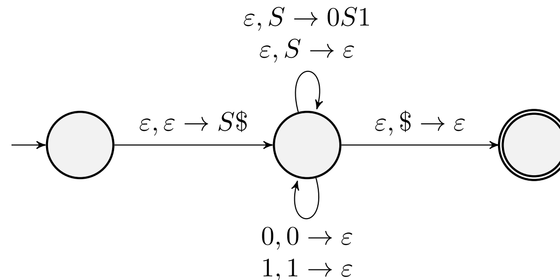


Proof. □

Example: Conside the grammar

$$S \to 0S1|\varepsilon$$

An equivalent PDA, using this construction, would be



Theorem 2. *Given a PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, we can design a CFG $G = (V, \Sigma, R, S)$ s.t. $L(G) = L(M)$.*

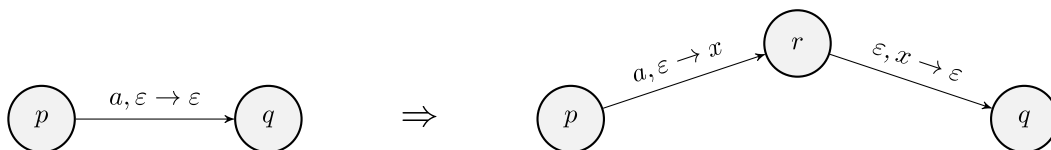
Proof. First, we assume the following about M, WLOG:

1. $|F| = 1$ (unique accept state)
2. Stack is empty when M accepts any string
3. In every transition, the PDA either pushes exactly one symbol or pops exactly one symbol, but never both.

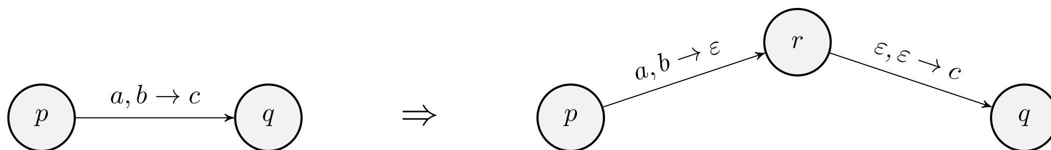
Exercise: Justify the claim that this assumption does not cause a loss of generality.

For any PDA, we can create an equivalent PDA with these properties:

1. Add ε-transitions from old accept states to new accept state, as for NFAs.
2. Convert old accept state to loop state that removes everything on the stack, then ε-transitions to new accept state. Need a new bottom-of-stack symbol added at beginning of PDA, below any bottom-of-stack symbol used by the original PDA.
3. Replace any non-conforming edges:
   (a) No push or pop:



   (b) Both push and pop:



Now, we can build our grammar. Consider any pair of states p and q. Add the variable A_{pq} to our grammar. A_{pq} will generate all strings which drive M from state $p\sqcup$ to state $q\sqcup$, where $\sqcup$ denotes "empty stack". That is, strings generated by A_{pq} get us from p to q without using anything already on the stack or leaving anything new behind. Specifically (and importantly), there might actually be something on the stack when we are in state p, but we do not interact with or depend on that, and end up in state q with the same stack contents as we had in p.

- $V = \{A_{pq}\}_{p,q \in Q}$.
- $S = A_{q_0 q_{accept}}$

- $R = \begin{cases} A_{pq} \to aA_{rs}b & \text{with } a,b \in \Sigma_\varepsilon \text{ and } \exists x \in \Gamma \text{ s.t. } \delta(p,a,\varepsilon) \text{ contains } (r,x) \text{ and } \delta(s,b,x) \text{ contains } (q,\varepsilon). \\ A_{pq} \to A_{pr}A_{rq} & \forall p,q \in Q \\ A_{pp} \to \varepsilon & \forall p \in Q \end{cases}$

Let us explain these rules: Since every step either pushes or pops, and we cannot touch anything on the stack when we are at state p, the first step must push, and the last must pop. We then have three cases:

1. If the stack is never empty between p and q, then the first step from p to some r will push some value x, and that value will stay until the last step from some s to q, which will pop x. In between, we must get from state r to state s without touching x, so we want a string which drives M from $r\sqcup$ to $s\sqcup$, that is, one in A_{rs}.
2. If the stack is empty at some point between p and q, say in state r, then we go from $p\sqcup$ to $r\sqcup$, then from $r\sqcup$ to $q\sqcup$, so we want substrings generated by A_{pr} and A_{rq}, respectively.
3. This is an inductive definition, so we need a base case, which is that ε takes us from a state to itself without touching the stack.

Claim 1. *$A_{pq} \overset{*}{\Rightarrow} x$ iff there is a path from $p\sqcup$ to $q\sqcup$ on input x.*

Proof. First, we prove the forward implication. Assume $A_{pq} \overset{*}{\Rightarrow} x$, and prove by strong induction on the number of steps in that derivation that there is a path from $p\sqcup$ to $q\sqcup$.

BC: Suppose $A_{pq} \Rightarrow x$ (derives in one step). Then $p = q$ and $x = \varepsilon$, since that is the only way to generate a string in one step in the grammar we defined, as all other rules produce another variable. It is true that there is a path from $p\sqcup$ to $p\sqcup$ on input ε, so the base case holds.

IH: Assume the claim holds for any string generated with no more than n derivations (rules expanded).

IS: We will show that if $A_{pq} \overset{*}{\Rightarrow} x$ in $n+1$ steps, there is a path from $p\sqcup$ to $q\sqcup$ on input x.

- If $A_{pq} \overset{*}{\Rightarrow} x$ in $n+1$ steps, the first step must be of one of the first two types above.
  1. First step is $A \Rightarrow aA_{rs}b$, where $A_{rs} \overset{*}{\Rightarrow} w$ in n steps and $x = awb$. By the inductive hypothesis, there is a path from $r\sqcup$ to $s\sqcup$ on input w, so we see that there is a path $p\sqcup \to r\sqcup \rightsquigarrow s\sqcup \to q\sqcup$ on input x, since rules of this type push on a and pop on b.
  2. First step is $A_{pq} \to A_{pr}A_{rq}$. For some y, z, we have $x = yz$ and $A_{pr} \overset{*}{\Rightarrow} y, A_{rq} \overset{*}{\Rightarrow} z$, in k and $n-k$ steps, respectively. By the inductive hypothesis, y drives M from $p\sqcup$ to $r\sqcup$, then z drives M from $r\sqcup$ to $q\sqcup$, so x drives M from $p\sqcup$ to $q\sqcup$.

Next, we prove the reverse implication, that if there is a path from $p\sqcup$ to $q\sqcup$ on input x, then $A_{pq} \overset{*}{\Rightarrow} x$. We do this by strong induction on the number of transitions on the path in the PDA.

BC: With 0 transitions, we have $x = \varepsilon$ and $p = q$. The grammar rule $A_{pp} \to \varepsilon$ proves the base case.

IH: Assume that for all strings x which drive M from $p\sqcup$ to $q\sqcup$ in no more than n transitions, $A_{pq} \overset{*}{\Rightarrow} w$.

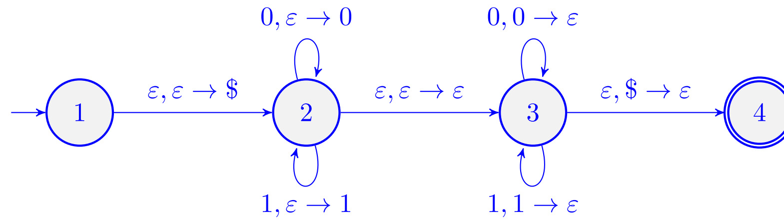
IS: Let x be any string that drives M from $p\sqcup$ to $q\sqcup$ in $n+1$ transitions.

1. If M, driven by x from $p\sqcup$, reads a and pushes u, going from state p to state r, and reads b and pops u while going from state s to state q, but never empties its stack between state r and state s, then let $x = awb$, for some $a, b \in \Sigma_\varepsilon$ and $w \in \Sigma^*$. Then w drives M from $r\sqcup$ to $s\sqcup$ in $n-1$ transitions, so by the IH, $A_{rs} \overset{*}{\Rightarrow} w$, and therefore $A_{pq} \to aA_{rs}b \overset{*}{\Rightarrow} awb = x$.
2. Otherwise, the stack is empty somewhere between p and q, say in state r. Then $x = yz$, where y drives M from $p\sqcup$ to $r\sqcup$ and z drives M from $r\sqcup$ to $q\sqcup$, each in no more than n transitions. By the inductive hypothesis, we then know that $A_{pr} \overset{*}{\Rightarrow} y$ and $A_{rq} \overset{*}{\Rightarrow} z$, so $A_{pq} \to A_{pr}A_{rq} \overset{*}{\Rightarrow} yz = x$.

□

□

Exercise: Give a CFG equivalent to the following PDA for even-length palindromes by building A_{pq} for each pair of states p and q.



We first list all possible variables A_{pq}, then determine what strings drive the machine from $p\sqcup$ to $q\sqcup$ for each variable.

1. Add $A_{pp} \to \varepsilon$ for each p.
2. Note that there are no edges in the PDA from higher-numbered states to lower-numbered states, so we can discard all such variables, as they generate no strings.
3. We cannot step forward from 1 or to 4 with empty stack before and after, unless we go from 1 to 4, since the \$ indicating empty stack is only added when leaving 1 and only removed when reaching 4, so all A_{1q} and A_{p4} are null except A_{14}.
4. For A_{14} (the first variable of interest), the stack will never be empty between states 1 and 4, since the empty stack symbol will be present. Thus, we add the first type of rule, noting that the $1 \to 2$ and $3 \to 4$ transitions both consume ε from the input, giving $A_{14} \to \varepsilon A_{23} \varepsilon$.
5. To get from state 2 to state 3, empty stack to empty stack, we must first loop at 2, then transition to 3, then loop at 3 to remove what we pushed in the first loop. The stack is never empty in between, so we use the first type of rule. We have two such rules, depdending on whether we push and pop a 0 or 1: $A_{23} \to 0A_{23}0 \mid 1A_{23}1$. We also see that we can go directly from state 2 to state 3 on ε, without pushing anything (note that this would be expanded to push then pop a dummy value, making it a rule of the first type, with middle portion reducing to ε since it is going from the dummy state to itself).
6. The start variable is A_{14}, since state 1 is the start state and state 4 is the accept state.

- $A_{11} \to \varepsilon$
- $A_{12} \to null$
- $A_{13} \to null$
- $A_{14} \to \varepsilon A_{23} \varepsilon = A_{23}$
- $A_{21} \to null$
- $A_{22} \to \varepsilon$
- $A_{23} \to 0A_{23}0 \mid 1A_{23}1 \mid \varepsilon$
- $A_{24} \to null$
- $A_{31} \to null$
- $A_{32} \to null$
- $A_{33} \to \varepsilon$
- $A_{34} \to null$
- $A_{41} \to null$
- $A_{42} \to null$

- $A_{43} \to null$
- $A_{44} \to \varepsilon$

If we write this without variables which generate nothing, and also discarding any unreachable variables, we obtain the following grammar:

$$A_{14} \to A_{23}$$
$$A_{23} \to 0A_{23}0 \mid 1A_{23}1 \mid \varepsilon$$