CS 4510 Automata and Complexity Sept 27

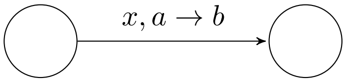
Lecture 9

Lecturer: Frederic Faulkner *Scribe(s):*

1 A New Computing Model

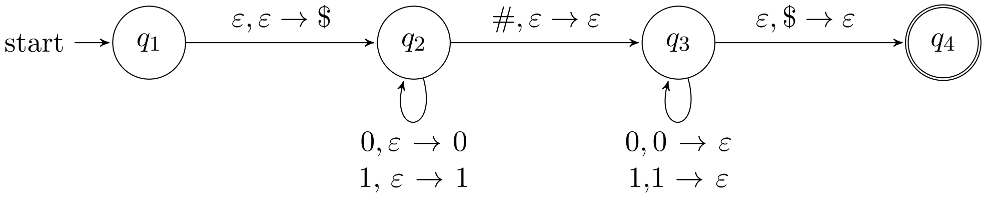
We now introduce a new model of computing called a Pushdown Automata (PDA). A PDA is effectively an NFA with a stack. This stack works just like other stacks you've used in your progamming classes: the NFA can push things on to the stack and pop things off. It can only access the topmost (most recently pushed) item on the stack. Unlike a normal NFA which decides which transition to take based on the next letter of the input, a PDA takes into account both the next input letter AND the top character of the stack.

Each transition in a PDA looks like so:



The leftmost character is the expected input character, the second character is to be popped off of the stack, and the final character is to be pushed onto the stack. This transition signifies "if reading an x while an a is on the stack, pop the a and replace it with b while taking this transition." Note that since PDA's are non-deterministic, any of a, b and x could be ε.

Consider the following machine.



The first transition, $\varepsilon, \varepsilon \to \$$, means "Do not read the input yet, do not pop anything off the stack, just push a \$ onto the stack." This is a very common starting transition for PDA's, since we have normally no way to check if the stack is empty. Now, if we see the \$ on the top of the stack, we know that we have cleared off every other character we put there, so the stack is "empty".

The loop on $q2$ pushes a 0 whenever we read a 0 and pushes a 1 whenever we read a 1. Conversely, the loop on q_3 expects a 0 on top of the stack whenever it reads a 0 and expects a 1 whenever it reads a 1. So what is the langauge of this PDA?

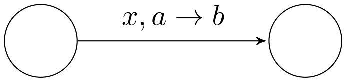
Consider its action on the string 011#110. The machine will first push a \$ and then transition to q_2. While it is there, it will push a 0 and two 1's onto the stack in that order, so that the stack looks like \$011. Then it will read the # and transition to state q_3. While it is here it will pop both 1's and then the 0, since that matches the remaining input, so the stack will just be \$. This allows us to transition to q_4. Since there is no input remaining to read and we are in a final state, we accept. It should be clear that the language of this PDA is $w\#w^R$, or the set of palindromes with # as the middle character.

1.1 Formalities

Definition 1 *Formally, a PDA is defined to be a 6-tuple* $(Q, \Sigma, \Gamma, q_0, F, \delta)$ *where* Q, q_0, F, Σ *have their usual interpretaions and*

- Γ *is the stack alphabet, or the set of characters you can push onto and pop off of the stack. A very common choice of* Γ *is just* $\Sigma \cup \{\$\}$*, but* Γ *can contain any symbols that you as the programmer find helpful as you construct your PDA.*
- δ*, the transition function is defined as* $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to P(Q \times \Gamma_\varepsilon)$*, where* L_ε *for any set* L *is defined to be* $L \cup \varepsilon$*.*

Again, recall that transition in a PDA looks like so:



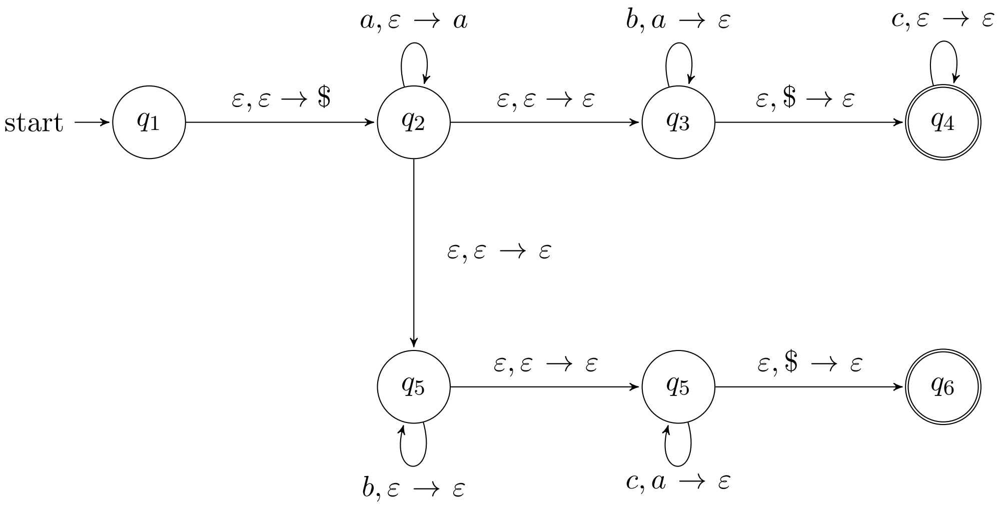
Just like in an NFA, x can be ε which signifies that that transition can be taken without reading the next character of the input yet. Additionally a and b can be ε:

- if neither a nor b is ε, then we replace a by b at the top of the stack
- if just a is ε, we just push b onto the stack on top of whatever was already there
- if just b is ε, we pop a off the stack
- if both a and b are ε, then that transition does not affect the stack at all.

1.2 Another Example

Consider the following language $L = \{a^i b^j c^k \mid i = j \text{ or } i = k\}$. How can use a stack to help us recognize this language? Our first thought might be to try storing the number of a's on the stack. We will then compare this to the number of b's, accept if they are equal, and if not compare the a's to the c's. However, there is an issue with this plan: once we have read all of the b's, the stack no longer has all the a's that we put in, since it popped some of them off as it read the b's. Thus, the remaining a's are useless to us, and we have no way of comparing the number of a's to the number of c's. We see that the "memory" of a PDA is effectively one-use! That is, we can store something in the stack and then recall it, but once we have recalled it once we forget it forever.

So how can we build a PDA for L? The trick is to take advantage of non-determinsim. As we run the PDA, we will simply guess whether we should compare a's to b's or to c's. See the diagram below, where the top branch corresponds to guessing that $i = j$, while the bottom branch handles the case $i = k$.


start
q_1
$\varepsilon, \varepsilon \to \$$
q_2
$a, \varepsilon \to a$
$\varepsilon, \varepsilon \to \varepsilon$
q_3
$b, a \to \varepsilon$
$\varepsilon, \$ \to \varepsilon$
q_4
$c, \varepsilon \to \varepsilon$
$\varepsilon, \varepsilon \to \varepsilon$
q_5
$b, \varepsilon \to \varepsilon$
$\varepsilon, \varepsilon \to \varepsilon$
q_5
$c, a \to \varepsilon$
$\varepsilon, \$ \to \varepsilon$
q_6

NOTE: The fact that PDA's are inherently non-deterministic is actually very important. Unlike for finite automata, for whom the deterministic and non-deterministic versions are equivalent, deterministic stack machines are significantly less powerful than non-deterministic ones. (The proof of this fact is outside of the scope of this course.) We will not discuss deterministic stack machines in this course, but the textbook has a section on them if you are interested.