1 Pushdown Automata (2.2)

We now introduce a new type of computational model called the **pushdown automata**, which is essentially like a nondeterministic finite automata but with an extra component called a **stack**. The stack provides additional memory beyond the finite amount available in the control, additionally allowing us to recognize some nonregular languages.

Note that pushdown automata are equivalent in power to context-free grammars, giving us the ability to decide which of the two we want to use to prove that a language is context-free.

1.1 The Idea

The idea is that, at each step:

1. **Transition** to a new state based on the current state, letter read, *and* the top letter of the stack.
2. Possibly push (or pop) a letter to (or from) the top of the stack.

We *accept* a string if there is **some** sequence of states and **some** sequence of stack contents which processes the entire input string and ends in an accepting state. We assume that the stack is empty at the beginning, and don't necessarily care if the stack is empty at the end.

1.2 Formal Definition

As implied, the formal definition of a pushdown automaton is similar to that of a finite automaton, except for the stack. The stack is a device containing symbols drawn from some alphabet, and the machine may use different alphabets for its input and its stack. Therefore, we need to specify both an input alphabet Σ and a stack alphabet Γ.

Recall that $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$; the same idea applies with Γ_ϵ. Now, the domain of δ is $Q \times \Sigma_\epsilon \times \Gamma_\epsilon$. The idea behind this domain is that, given the current state $q \in Q$, next input symbol read $\sigma \in \Sigma_\epsilon$, and top symbol of the stack $\gamma \in \Gamma_\epsilon$, the machine can decide the next move. As usual, if $\sigma = \epsilon$, then the machine won't read a symbol from the input. Likewise, if $\gamma = \epsilon$, then the machine won't read a symbol from the stack.

We now consider the range of the δ function. It's possible that it can enter a new state and possibly write a symbol on top of the stack. This is indicated by some member of $Q \times \Gamma_\epsilon$.

Definition 1.1

A **pushdown automaton** is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, and F are all finite sets, and

1. Q is the set of states.
2. Σ is the input alphabet.
3. Γ is the stack alphabet.
4. $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$ is the transition function.
5. $q_0 \in Q$ is the start state.
6. $F \subseteq Q$ is the set of accept states.

Remark: We write $a, b \mapsto c$ to signify that, when the machine is reading an a from the input, it may replace the symbol b on the top of the stack with a c; in other words, pop b and push c.

- If $a = \epsilon$, then the machine may make this transition without reading any symbol from the input.

- If $b = \epsilon$, then the machine may make this transition without readng any symbols from the stack, but it will push c onto the stack.
- If $c = \epsilon$, then the machine will pop b from the stack but not push anything to the stack.

Note: The formal definition of a PDA contains no explicit mechanism to allow the PDA to test for an empty stack. So, one trick that we can do is to initially push a special symbol `$` on the stack. Then, if it ever sees the `$` again, then it knows that the stack is effectively empty.

1.3 Computation of a Pushdown Automaton

A pushdown automaton $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ accepts input w if w can be written as $w = w_1 w_2 \dots w_m$, where each $w_i \in \Sigma_\epsilon$ and the sequences of starts $r_0, r_1, \dots, r_m \in Q$ and strings $s_0, s_1, \dots, s_m \in \Gamma^*$ exist that satisfy the following three conditions.

1. $r_0 = q_0$ and $s_0 = \epsilon$, which signifies that M starts out properly, in the start state, and with an empty stack.
2. For $i = 0, \dots, m-1$¡ we have $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, where $a_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Gamma_\epsilon$ and $t \in \Gamma^*$. This condition states that M moves properly according to the state, stack, and next input symbol.
3. $r_m \in F$. This condition states that the accept state occurs at the input end.

1.4 Context-Free

Theorem 1.1

A language is context free if and only if some pushdown automaton recognizes it.

1.4.1 Example 1: Designing a PDA

Consider the following language $L = \{\mathtt{0}^i\mathtt{1}^{i+1} \mid i \geq 0\}$.

1. Design a CFG that generates L.

   We can define a CFG G where the rules are
   $$S \mapsto T\mathtt{1} | \mathtt{0}S\mathtt{1}$$
   $$T \mapsto \epsilon$$
   The idea is that S can recursively map to $\mathtt{0}S\mathtt{1}$. After enough times, it can then map to $T\mathtt{1}$, which adds one more `1`.

2. Consider the notation $\epsilon, \epsilon \mapsto \mathtt{\$}$. What does this mean?

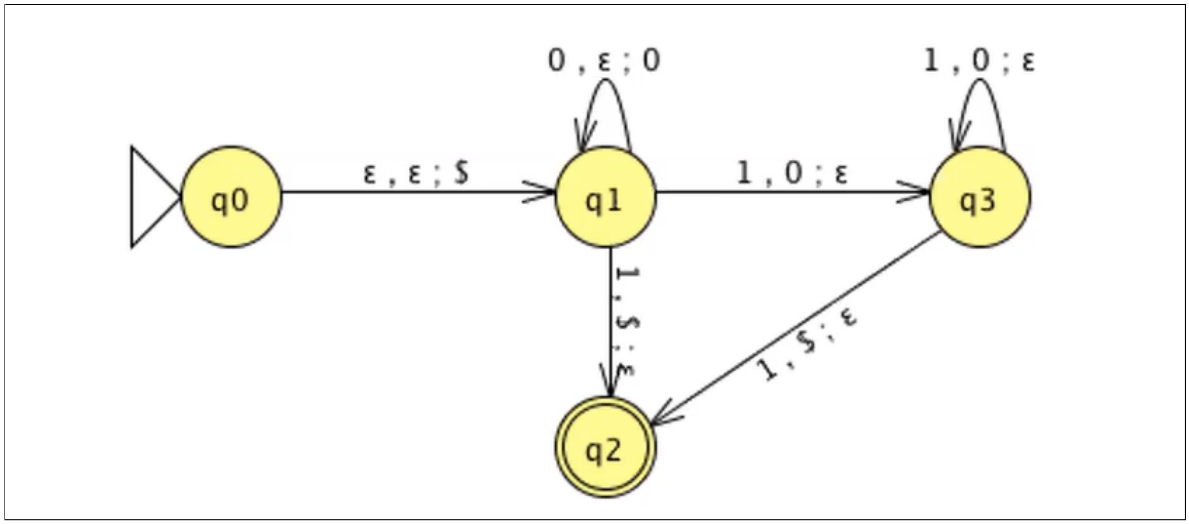
   Without reading any input, and without popping any symbols from the stack, we push `$` on top of the stack.

   Remark: This is commonly used from the initial state (at the start of computation) to record the top of the stack with a special symbol. In other words, if the stack becomes empty, then we will know.

3. Provide a rough description of how you can make a PDA that accepts upon receiving any strings that are in this language.

The idea for a PDA is that we need to read the symbols from the input. As each `0` is read, we push it onto the stack. As soon as `1`'s are seen, we pop a `0` for each `1` that is read. If the stack becomes empty and there is exactly one `1` left to read, then we read that `1` and accept the input. If the stack becomes empty and there are either zero or more than one `1`'s left to read, *or* if the `1`'s are finished while the stack still contains `0`'s, *or* if any `0`'s appear in the input following the `1`'s, then reject the input.

4. Design a PDA that does exactly what you described in the previous step.



5. Describe how your PDA works with the input `00111`.

- For our initial step, we start at q_0 and our stack can be described as `[]`.
- Looking at the only transition arrow, note that we can transition without reading any input ($\epsilon, \epsilon \mapsto \$$). Upon transitioning, we don't pop anything from the stack, but we do push the special `$` onto the stack. So, we can describe the machine like so:

```
State:  q1
Stack:  [$]     Top
Input:  00111
        ^
```

- At q_1, we have three transition arrows; one that requires a `0` (without reading and popping anything from the stack), and two that requires a `1` (one which requires the `$` to be at the top of the stack, and another that requires a `0` to be on top of the stack). As the (first and) next symbol in our input is `0`, we read that in and transition to q_1 again while also pushing `0` onto the stack.

```
State:  q1
Stack:  [$, 0]     Top
Input:  00111
         ^         Next to read
```

- We repeat the previous step since we need to read in a `0`.

```
State:  q1
Stack:  [$, 0, 0]     Top
Input:  00111
```

```
                ^           Next to read
```

- At this point, our next input symbol is `1`. As we're at q_1 still, we now need to consider the two transition arrows that can read in a `1`. One transition arrow requires a `0` to be on top of the stack, and another requires a `$`. So, we go with the one that requires a `0` to be on top of the stack since that's what we have in the stack. So, we transition to q_3, popping a `0` from the stack.

```
State:  q2
Stack:  [$, 0]     Top
Input:  00111
           ^       Next to read
```

- We are now at state q_3. The next input symbol we need to read in is the `1` (the center one), and we still have another `0` on top of the stack. So, we take the transition arrow $\texttt{1}, \texttt{0} \mapsto \epsilon$. After transitioning back to q_3, we pop `0` from the stack.

```
State:  q2
Stack:  [$]     Top
Input:  00111
            ^   Next to read
```

- Now that we're still at q_3, we still need to read in `1` (the last one). However, the top of the stack is a `$`, implying that we're at the end of the stack. We note that there is a transition arrow $\texttt{1}, \$ \mapsto \epsilon$, we take that transition arrow. So, we transition to q_2 and pop `$` from the stack.

```
State:  q2
Stack:  []     Top
Input:  00111
             ^ Next to read
```

- There is nothing left to read, and since we're at q_2, we accept.

For a summary of what just happened, observe the below table.

Input	q	**Stack** (Top is Right)
	q_0	`[]`
	q_1	`[$]`
`0`	q_1	`[$, 0]`
`0`	q_1	`[$, 0, 0]`
`1`	q_3	`[$, 0]`
`1`	q_3	`[$]`
`1`	q_2	`[]`

1.4.2 Example 2: PDA vs. CFGs

Consider the following language $L = \{\texttt{a}^i\texttt{b}^j\texttt{c}^k \mid i = j \text{ or } i = k \text{ with } i, j, k \geq 0\}$.

1. Which of the following strings are **not** in L?
   - `b`
   - `abc`
   - `abbcc`

- `aabcc`

The answer is `abbcc`. This says that $i = 1$, $j = k = 2$, but this violates the definition that $i = j$ or $i = k$ since $i \neq j$ and $i \neq k$.

2. Create a CFG that generates L.

The idea is that

$$\begin{aligned} L &= \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i = j \text{ or } i = k \text{ with } i, j, k \geq 0\} \\ &= \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i = j \text{ with } i, j, k \geq 0\} \cup \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i = k \text{ with } i, j, k \geq 0\} \end{aligned}$$

So, we can find the CFG of each individual language and then combine them to form the CFG of interest.

- $L_1 = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i = j \text{ with } i, j, k \geq 0\}$. The CFG can be defined by

$$\begin{aligned} A &\mapsto A\mathtt{c} \mid B \\ B &\mapsto \mathtt{a}B\mathtt{b} \mid \epsilon \end{aligned}$$

- $L_2 = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i = k \text{ with } i, j, k \geq 0\}$. The CFG can be defined by

$$\begin{aligned} C &\mapsto \mathtt{a}C\mathtt{c} \mid D \\ D &\mapsto D\mathtt{b} \mid \epsilon \end{aligned}$$

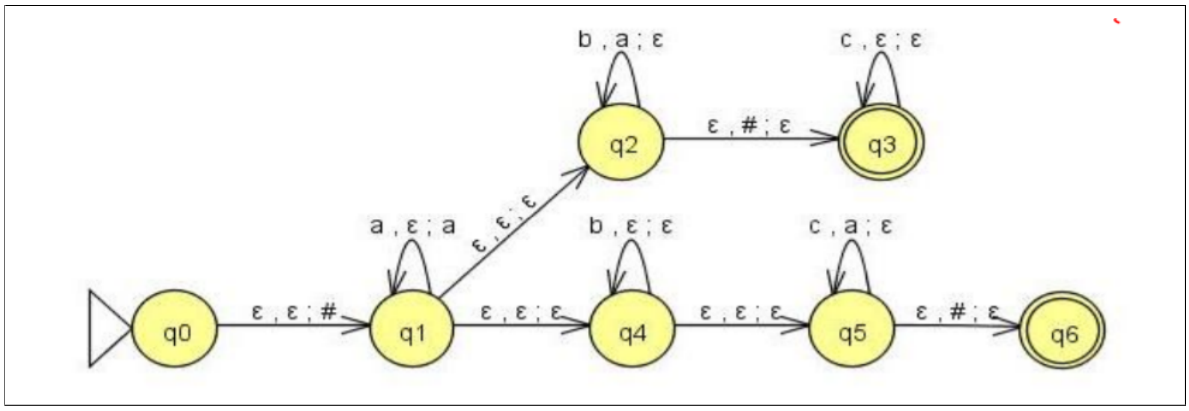
Combining these gives us

$$\begin{aligned} S &\mapsto A \mid C \\ A &\mapsto A\mathtt{c} \mid B \\ B &\mapsto \mathtt{a}B\mathtt{b} \mid \epsilon \\ C &\mapsto \mathtt{a}C\mathtt{c} \mid D \\ D &\mapsto D\mathtt{b} \mid \epsilon \end{aligned}$$

3. Give an informal description of a PDA that accepts all strings in this language. *Hint:* Consider what information you need to track, the amount of memory you need, and whether non-determinism is needed.

- The PDA pushes a `$` to indicate the top of the stack. Then, it starts reading `a`'s, pushing each one onto the top of the stack.
- The PDA then guesses when it has reached the end of the `a`'s and whether to match the number of `a`'s to the number of `b`'s or the number of `c`'s.
- If trying to match number of `b`'s with the number of `a`'s, the PDA pops off `a`'s for each `b` that is read in. If there are more `a`'s on the stack but no more `b`'s being read, then we reject. When the end of the stack `$` is reached, the number of `a`'s matches the number of `b`'s. If this is the end of the input or if any number of `c`'s is read at this point, accept. Otherwise, reject.
- If trying to match the number of `c`'s with the number of `a`'s, first read any number of `b`'s without changing the stack content. Then, nondeterministically guess when to start reading the `c`'s. For each `c` read, pop one `a` off the stack. When the end of the stack `$` is reached, the number of `a`'s and `c`'s match.

4. Design a PDA that does exactly what you described in the previous step.



1.5 Conventions for PDAs

- We can *test for the end of the stack*, without providing details. We can always push the end-of-stack symbol `$` at the start.
- We can *test for the end of the input* without providing details. We can transform the PDA to one where accepting states are only those reachable when there are no more input symbols.
- We don't always need to provide a state transition diagram.