Push-down automata (PDAs) are essentially limited versions of Turing machines. We will use them to efficiently determine if a CFG $G$ can generate a parse tree for a sentence $x$. We only consider deterministic PDAs; that is, for any given configuration at most one move, and perhaps no move, is possible.

We will define a PDA $P$ to have:

- a set of $m$ distinct states $s_1, \ldots, s_m$
  - $s_1$ is the initial state; $s_m$ is the accept state
- an input alphabet of $l$ distinct symbols $a_1, \ldots, a_l$
  - $a_1$ is the right-end marker $\|$  
- a stack alphabet of $k$ distinct symbols $A_1, \ldots, A_k$
  - $A_1$ is the initial stack symbol $\pm$
- rules of the form $(s, a_i, A_j) \rightarrow (s', \alpha)$ where $s$ and $s'$ are states, $a_i$ is a single input symbol, $A_j$ is a single stack symbol and $\alpha$ is a sequence of stack symbols or the word “pop”.

where $l \geq 2; k, m \geq 1$. We impose a restriction that no two rules have the same left-hand side (this is the determinism condition). Every time a rule is applied, the top stack symbol $A_j$ is removed to be read, and based on it, the input symbol $a_i$ under the input head, and the current state $s$, the PDA will enter state $s'$, push $\alpha$ onto the stack (or add nothing if $\alpha = \text{"pop"}$), and move the input head one space to the right.

A legal input to $P$ would be a finite sequence $x = x_1x_2\ldots x_n$, where each $x_i$ is one of $a_2, \ldots, a_l$ (the input alphabet except the right-end marker; repeats of input symbols are permitted in $x$).

If $P$ had rules $(s_1, x_1, \pm) \rightarrow (s_2, A_7A_5)$ and $(s_2, x_2, A_7) \rightarrow (s_{15}, \text{pop})$, then the first three configurations of $P$ on input $x$ would be as follows, where the first configuration shows the initial configuration in which $P$ would start:

$$
\begin{align*}
\text{Initial Configuration} &:\quad \text{x}_1 \quad \text{x}_2 \quad \text{x}_3 \quad \ldots \quad \text{x}_n \quad \|=  \\
\quad \text{s}_1 & \\
\text{Configuration 1} &:\quad \text{x}_1 \quad \text{x}_2 \quad \text{x}_3 \quad \ldots \quad \text{x}_n \quad |  \\
\quad \text{x}_1 & \\
\quad \text{Configuration 2} &:\quad \text{x}_1 \quad \text{x}_2 \quad \text{x}_3 \quad \ldots \quad \text{x}_n \quad |  \\
\quad \text{A}_7 & \\
\quad \text{A}_5 & \\
\text{Configuration 3} &:\quad \text{x}_1 \quad \text{x}_2 \quad \text{x}_3 \quad \ldots \quad \text{x}_n \quad |  \\
\quad \text{x}_1 & \\
\quad \text{s}_{15} & \\
\quad \text{A}_5 & \\
\end{align*}
$$

$P$ accepts $x$ if it can start in the initial configuration corresponding to $x$ and, following its rules, have the input head fall off the input tape while changing to its accept state. If it would halt in any other configuration, such as getting stuck somewhere on the tape because no rule applies or falling off of the end of the tape but not in the accept state, it does not accept $x$.  

---

1
Example PDAs

PDA One

- States: count-a, match-b, matched
- Initial state: count-a
- Accept state: matched
- Input symbols: ⊾, a, b
- Stack symbols: ±, A, M
- Moves:
  1. (count-a, a, ±) → (count-a, AM)
  2. (count-a, a, A) → (count-a, AA)
  3. (count-a, b, A) → (match-b, pop)
  4. (match-b, b, A) → (match-b, pop)
  5. (match-b, ⊾, M) → (matched, M)

PDA Two - trying to “optimize” PDA One

- States: count-a
- Initial state: count-a
- Accept state: count-a
- Input symbols: ⊾, a, b
- Stack symbols: ±, A
- Moves:
  1. (count-a, a, ±) → (count-a, A±)
  2. (count-a, a, A) → (count-a, AA)
  3. (count-a, b, A) → (count-a, pop)
  4. (count-a, ⊾, ±) → (count-a, ±)