Main topics of the week:
- Context Free Grammars and parse trees
- Regular Languages and CFGs
- Chomsky Normal Form for CFGs
- Pushdown Automata Definition and examples
- Construction of PDA from a CFG
- Construction of CFG from a PDA

CFGs for Regular Languages.
A CFG for a regular language is easy if you have the DFA for the language. To create the CFG from the DFA, just make a variable for each state and have a rule of the form \( R_i \rightarrow aR_j \) if there is a transition by input ‘a’ from \( R_i \) to \( R_j \). Have a rule that takes each variable corresponding to an accept state to \( \varepsilon \).

A right-linear grammar is a grammar in which all productions are of the form \( A \rightarrow xB \) where \( x \) is a terminal
or \( A \rightarrow \varepsilon \). The construction above shows that a DFA can be used to create a right-linear grammar. A left-linear grammar has a similar definition. A regular grammar is a right or left linear grammar, and corresponds to a regular language.

CFG from DFA for regular language \( \{ w \in \{0,1\}^* \mid \text{every 0 is followed by a 1} \} \)

\[
\begin{align*}
S & \rightarrow 0A \mid 1S \mid \varepsilon \\
A & \rightarrow 0B \mid 1S \\
B & \rightarrow 0B \mid 1B
\end{align*}
\]

Simplified to:
\[
S \rightarrow 01S \mid 1S \mid \varepsilon
\]
Example of Chomsky normalization.

This example is for a grammar which produces balanced parentheses. The right side at each stage shows the way the grammar has been changed. The highlighted text indicates what is being eliminated on the left and what are the corresponding additions on the right.

\[
\begin{align*}
S &\rightarrow (S) | SS | \varepsilon \\
\end{align*}
\]

First we add a new start symbol.

\[
\begin{align*}
S &\rightarrow (S) | SS | \varepsilon \\
S_0 &\rightarrow S \\
S &\rightarrow (S) | SS | \varepsilon \\
\end{align*}
\]

Then we remove the \( \varepsilon \) rule \( S \rightarrow \varepsilon \).

\[
\begin{align*}
S_0 &\rightarrow S \\
S &\rightarrow (S) | SS | \varepsilon \\
S &\rightarrow (S) | SS | \varepsilon \\
S &\rightarrow (S) | SS | ( ) | S \\
\end{align*}
\]

Now we remove the unit rules \( S \rightarrow S \) and \( S_0 \rightarrow S \).

\[
\begin{align*}
S_0 &\rightarrow S | \varepsilon \\
S &\rightarrow (S) | SS | ( ) | S \\
S &\rightarrow (S) | SS | ( ) \\
\end{align*}
\]

Individual rules are added for terminals appearing in strings of length \( \geq 2 \).

\[
\begin{align*}
S_0 &\rightarrow (S) | SS | ( ) | S \\
S &\rightarrow (S) | SS | ( ) \\
S &\rightarrow (S) | SS | LR \\
L &\rightarrow ( \\
R &\rightarrow ) \\
\end{align*}
\]

Finally, reduce all strings of length \( \geq 3 \), adding new variables as necessary.

\[
\begin{align*}
S_0 &\rightarrow LSR | SS | LR | \varepsilon \\
S &\rightarrow LSR | SS | LR \\
L &\rightarrow ( \\
R &\rightarrow ) \\
\end{align*}
\]
This is another example of converting a CFG to Chomsky normal form. First we add a new start symbol:

\[
S \to aSa \mid AB \mid a
\]
\[
A \to CC \mid aaA \mid \varepsilon
\]
\[
B \to bBb \mid \varepsilon
\]
\[
C \to cCc \mid \varepsilon
\]

Then we remove \( \varepsilon \) rules for A, B, C.

\[
S_0 \to S \quad S \to aSa \mid AB \mid a
\]
\[
A \to CC \mid aaA \mid \varepsilon \quad A \to CC \mid aaA \mid \varepsilon
\]
\[
B \to bBb \mid \varepsilon \quad B \to bBb \mid bb
\]
\[
C \to cCc \mid \varepsilon \quad C \to cCc \mid cc
\]

Then we remove unit rules \( S \to B, S \to A, A \to C \).

\[
S_0 \to S \quad S \to aSa \mid AB \mid a \mid bBb \mid bb \mid CC \mid aaA \mid aaC
\]
\[
A \to CC \mid aaA \mid aa \mid cCc \mid cc
\]
\[
B \to bBb \mid bb
\]
\[
C \to cCc \mid cc
\]

And now remove the newly appearing unit rules \( S \to C \) and \( S_0 \to S \).

\[
S_0 \to S
\]
\[
S \to aSa \mid AB \mid a \mid bBb \mid bb \mid CC \mid aaA \mid aaC
\]
\[
A \to CC \mid aaA \mid aa \mid cCc \mid cc
\]
\[
B \to bBb \mid bb
\]
\[
C \to cCc \mid cc
\]

Now convert the remaining rules to the desired form.

\[
S_0 \to aSa \mid AB \mid a \mid bBb \mid bb \mid CC \mid aaA \mid aaC
\]
\[
S \to aSa \mid AB \mid a \mid bBb \mid bb \mid CC \mid aaA \mid aaC
\]
\[
A \to CC \mid aaA \mid aa \mid cCc \mid cc
\]
\[
B \to bBb \mid bb
\]
\[
C \to cCc \mid cc
\]
\[
U_1 \to a
\]
**Pushdown Automata Examples**

Let $A$ be the language $\{ 0^n 1^n \mid n \geq 0 \}$, which we know is not a regular language. We show a pushdown automaton that recognizes $A$. For the formal specification, we need a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$:

1. $Q = \{ q_1, q_2, q_3, q_4 \}$
2. $\Sigma = \{ 0, 1 \}$
3. $\Gamma = \{ 0, \$ \}$
4. $F = \{ q_1, q_4 \}$

For $\delta$ we could use a 3-dimensional table for $\delta$, but instead we’ll give the non-$\emptyset$ mappings:

- $\delta(q_1, \epsilon, \epsilon) = \{(q_2, \$)\}$
- $\delta(q_2, 0, \epsilon) = \{(q_2, 0)\}$
- $\delta(q_2, 1, 0) = \{(q_3, \epsilon)\}$
- $\delta(q_3, 1, 0) = \{(q_3, \epsilon)\}$
- $\delta(q_3, \epsilon, \$) = \{(q_4, \epsilon)\}$

Notice that the first transition is really just for initialization since it happens with no input, and pushes the $\$ on the stack. The $\$ is basically going to be used as a marker for the bottom of the stack. This also advances us to the second state. Now when we see $0$, we stay in the second state, but push $0$ on the stack. If we see $1$ on input when we have $0$ at the top of the stack, we pop it and advance to the third state. Note that if we saw a $1$ first, we would halt as this transition is not defined since the stack would have $\$ on the top. Once in the third state, with each $1$ and still a $0$ on the stack, we pop it but stay in the third state. When we see $\$ on the stack (no input), we pop it and advance to the accept state. Note that the accept state has no transitions, so we cannot have any more input in order to be accepted. So we’re stuck if we see the $\$ and haven’t seen any $1$’s, or we still have input. We’re also stuck if we see any $0$’s after we have seen a $1$. And most importantly, the number of $0$’s and $1$’s must be the same.

We can also draw state diagrams for PDAs. In addition to our usual conventions for drawing, we now add to our arrow label to indicate what is happening to the stack. This will just be the mapping as given in the definition, from $\Gamma_e$ to $\Gamma_e$. A mapping $\epsilon \rightarrow b$ means to push $b$ on the stack, $a \rightarrow \epsilon$ means to pop $a$ from the stack, $\epsilon \rightarrow \epsilon$ means to do nothing to the stack, and $a \rightarrow b$ means to pop $a$ and push $b$.

Here is the diagram for the PDA for $0^n 1^n$ just described formally. Note that the start state is an accept state, and this allows empty strings. We see the transitions as discussed above, with the first being an initialization where $\$ is pushed on the stack. Then we loop.
on 0’s, pushing 0 on the stack for each input 0. 1’s take us to the next state and also pop a 0, which must be on the stack, else we would halt. We loop on 1’s and popping 0’s until the stack is empty (i.e., has our $ marker).

Here’s another example, the language of all strings that have an equal number of 0s and 1s. In this case, we’ll use the stack to match the 0s and 1s as we go along, which means we will be pushing 0s on until we see 1s at which point we’ll pop 0s for each 1. When the stack is empty and we see 1s, we’ll push 1s until we see 0s to match. The trick is noticing when the stack is empty so that we can know to switch to pushing. We do this by first initializing the stack as before (the transition to q2) with the marker ‘$’. From this state, if we see a 0, we push 0s (state q3) and also in this state, whenever we see 1s, we pop the 0s. If the stack ever gets empty, we return to q2 and preserve our marker. We use state q4 similarly to push 1s and match 0s against them. What makes this work is that whenever the stack is empty, we return to our neutral state so we can decide whether to push 1s or 0s, depending on the input. Notice that we are not requiring equal consecutive numbers of 0s and 1s, just that they ultimately balance.

In fact, we could have eliminated the last state q5 by having the transition there go back to the start state instead. Essentially, this would mean that we would be in the accept state whenever the 0s and 1s were balanced, and if there is no more input, the string would be accepted. (If we had done this in the previous example, it would have extended the language to be all strings where 0s were followed by an equal number of 1s.)
One more example: we used the pumping lemma to show that the palindromes are not regular. Let’s now construct a PDA to recognize palindromes. Notice that we really need nondeterminism here to “guess” when we are at the middle of the string. That is, we push 0s and 1s on the stack, then guess to begin popping as we match to the remaining input. Without nondeterminism to make this guess, this would be very hard to come up with a PDA.

Example of constructing a PDA from a CFG.

Here is a concrete example of using this construction of Lemma 2.13 on a CFL. We try this for the simple expression language defined by the grammar we have seen before:

- \( E \rightarrow E + T \mid T \)
- \( T \rightarrow T * F \mid F \)
- \( F \rightarrow ( E ) \mid a \)

Here the variables are \( E, T, \) and \( F, \) and the terminals are \( a, +, *, (, \) and \( ). \) There are six rules in all. The following diagram shows the PDA constructed for this language. We have shown the longhand transitions for the non unit rules using intermediate states. Note that the symbols are pushed on the stack in the reverse order from how they occur in the rule. Three of the rules are unit rules, so require no intermediate state. Finally, there are five terminals, so five corresponding transitions on matching input.
Rule: $E \rightarrow E + T$

Rule: $T \rightarrow T^*F$

Rule: $F \rightarrow (E)$

Unit Rules:
- $E \rightarrow T$
- $T \rightarrow F$
- $T \rightarrow a$
- $E \rightarrow T$
- $E \rightarrow E$
- $F \rightarrow T$
- $F \rightarrow F$
- $F \rightarrow a$
- $F \rightarrow a$
- $F \rightarrow a$
- $F \rightarrow a$

Terminals:
- $a, a \rightarrow \varepsilon$
- $+, + \rightarrow \varepsilon$
- $*, * \rightarrow \varepsilon$
- $(), ( \rightarrow \varepsilon$
- $), ) \rightarrow \varepsilon$

PDA to recognize simple expressions
Constructing a CFG from a PDA.

Recall that we make some simplifying assumptions about our PDA:
1. There is a single accept state.
2. The stack is emptied before accepting.
3. Each transition either pushes or pops a symbol from the stack, but not both.

The gist of our grammar construction is to define a variable for each pair of states such that the variable generates exactly the strings that move the PDA between the two states, leaving the stack in the same condition (i.e., strings that move between P between the states starting and ending with an empty stack.) Suppose that we can construct such a grammar. If the pair is the start state and accept state, then the strings that move between them with empty stack are exactly those strings accepted by P (remember that we guaranteed that the stack is emptied before acceptance). The reason for talking about an arbitrary pair of states is that we will prove this equivalence between generated strings and moving through P with empty state by induction. The goal is to get it to be true for the start and accept state, and we accomplish this by an induction proof that shows it is true for every pair.

We define our grammar G as follows, where $P = \{Q, \Sigma, \Gamma, \delta, q_{start}, \{q_{accept}\}\}$:

1. The variables of G are $\{A_{pq} | p,q \in Q\}$ [one variable for each state pair]
2. The start variable of G is $A_{start,accept}$ [this will show that G matches P]
3. For each $p,q,r,s \in Q$ and $t \in \Gamma$ and $a,b \in \Sigma$ where $(r,t) \in \delta(p,a,\varepsilon)$ and $(q,\varepsilon) \in \delta(s,b,t)$, we have the rule $A_{pq} \rightarrow aA_{rs}b$ [if there are states where we initially push a symbol from p, and pop that symbol to get to q, add this to the grammar]
4. For every $p,q,r \in Q$, we have the rule $A_{pq} \rightarrow A_{pr}A_{rq}$ [rule for every indirect path – but adding no terminals]
5. For every $p \in Q$, we have the rule $A_{pp} \rightarrow \varepsilon$ [null termination for the diagonal pairs]

The following pictures might help to see these rules.
**First Claim:** If $A_{pq}$ generates the string $x$, then $x$ can bring $P$ from $p$ with empty stack to $q$ with empty stack.

**Proof:** This proof will proceed by induction on the number of steps in the derivation:

If the derivation has one step, then there is a single rule that substitutes just terminals. In our definition of $G$, the only such rules are the ones from (5), i.e., $A_{pp} \rightarrow \varepsilon$. Thus the generated string must be $\varepsilon$, and $\varepsilon$ certainly can take $p$ with empty stack to $p$ with empty stack. (It’s just the transition $\varepsilon, \varepsilon \rightarrow \varepsilon$ which has no effect.)

For the induction step, assume the claim hold for any derivation with $k$ or fewer steps ($k \geq 1$). Suppose we have a derivation with $k+1$ steps. Since our rules have just two forms (in 3 and 4), we’ll consider each of these possibilities:

If the first step uses a rule $A_{pq} \rightarrow aA_{rs}b$, think about the portion of $x$ minus the $a$ and $b$, i.e., $x=ayb$. This portion, called $y$, is generated by $A_{rs}$ with $k$ steps (all but the first step in the derivation for $x$). By our induction assumption, this means that $y$ must take $r$ with empty stack to $s$ with empty stack. By our definition of $G$, this means that if we start with $x=ayb$ in $p$ with empty stack, we can transition to $r$ with input $a$ and pushing $t$, then transition somehow to $s$ with input $y$, leaving the stack alone, so that $t$ is still on the stack when we transition from $s$ to $q$ on input $b$, popping $t$, and thus $x$ gets us to $q$ with empty stack.

**Case 1:** First rule is $A_{pq} \rightarrow aA_{rs}b$. Write $x = ayb$, where $A_{rs}$ produces $y$
For the second case, where the first step is a rule $A_{pq} \rightarrow A_{pr} A_{rq}$, we must have $x = yz$, where $y$ and $z$ are the portions generated by the two variables on the right. Since the total steps in the derivation is $k+1$, each of $A_{pr}$ and $A_{rq}$ can have at most $k$ steps. So again by the induction hypothesis, $y$ goes from $p$ to $r$ with empty stack, and $z$ goes from $r$ to $q$ with empty stack. Obviously, then $x$ goes from $p$ to $q$ with empty stack.

This completes the induction proof, so our first claim is now proved.

**Second Claim**: If $x$ brings $P$ from $p$ with empty stack to $q$ with empty stack, then $A_{pq}$ generates $x$.

**Proof**: Again we will use induction, this time on the number of steps in the computation taking place in $P$.

For the basis, consider the case where the computation has zero steps. This means it starts and ends in the same state $p$. So we need to show that $A_{pp}$ generates $x$. Since there are zero steps in the computation, no input is read, hence $x$ must be the empty string $\varepsilon$. By our clever definition of $G$ (rule 5), $A_{pp}$ just happens to generate $\varepsilon$.

For the induction step, we assume that the claim holds if the number of steps in the computation is $k$ or fewer, and suppose we have a computation that takes $k+1$ steps. There are two possibilities: the stack may be empty only at the beginning and end, or it may be empty somewhere in between as well.

For the first case, suppose $t$ is the symbol pushed on the first transition of the computation. Then since this is the case where the stack is never empty, and we know that $P$ cannot push and pop in one step, $t$ must still be there at the last transition. You can see where we’re going – let $a$ and $b$ be the input for these first and last transitions, $r$ the second state, and $s$ the next to last state. Then we appeal to our definition of $G$ to know it has the rule $A_{pq} \rightarrow aA_{rs}b$. We can realize $x$ as aby, and know that $y$ brings $r$ to $s$, leaving $t$ on the stack. So of course $y$ brings $r$ with empty stack to $s$ with empty stack. Since we are taking out the first and last steps, the number of steps in this computation is certainly no more than $k$, hence $A_{rs}$ generates $y$ by the induction hypothesis. Combine this with the rule above, and we get a derivation for $x$.  

---

**Case 2**: First step is $A_{pq} \rightarrow A_{pr} A_{rq}$. $x = yz$ where $A_{pr}$ produces $y$, $A_{rq}$ produces $z$. 

---

**Diagram:**

```
  p  y  r  z  q
  |     |     |
  ▼    ▼    ▼
  steps ≤ k, so y goes from p to r with empty stack [by induction hypothesis]
  steps ≤ k, so z goes from r to q with empty stack [by induction hypothesis]
```
For the second case where the stack does become empty in the middle, let \( r \) be the state where that happens, and \( y \) and \( z \) be the corresponding breakdown of the input \( x \). The number of steps on either side of \( r \) is no more than \( k \), so the induction assumption says there must be derivations such that \( A_{pr} \) generates \( y \) and \( A_{rq} \) generates \( z \). Since we were clever enough to include all pair rules, we know the rule \( A_{pq} \rightarrow A_{pr}A_{rq} \) is in \( G \), thus we have our derivation of \( x \).

So where are we? We have proved our two claims about pairs of states and the grammar derivations. In particular for the start and accept states this means that we have proved our theorem about the equivalence of pushdown automata and CFLs.