MODULE III:

1. **Context Free Grammar**
2. **Minimization of Context Free Grammar**
3. **Chomsky Normal Form**
4. **Pumping Lemma for Context-Free Languages**
5. **Pushdown Automata (PDA)**

1. Ambiguity in Context Free Grammar

- **Definition:** Let G be a CFG. Then G is said to be ambiguous if there exists an x in L(G) with >1 leftmost derivations. Equivalently, G is said to be ambiguous if there exists an x in L(G) with >1 parse trees, or >1 rightmost derivations.
- Note: Given a CFL L, there may be more than one CFG G with L = L(G). Some ambiguous and some not.
- Definition: Let L be a CFL. If every CFG G with L = L(G) is ambiguous, then L is inherently ambiguous.
- **Example:** Consider the string aaab and the preceding grammar.
- The string has two left-most derivations, and therefore has two distinct parse trees and is ambiguous .

1.1 Eliminations of Useless Symbols

- **Definition:**
  Let $G = (V, T, S, P)$ be a context-free grammar. A variable $A \in V$ is said to be useful if and only if there is at least one $w \in L(G)$ such that

  $$S \Rightarrow^* xAy \Rightarrow^* w$$
  with $x, y \in (V \cup T)^*$.

  In words, a variable is useful if and only if it occurs in at least on derivation. A variable that is not useful is called useless. A production is useless if it involves any useless variable
- For a grammar with productions

$S \rightarrow aSb \mid \lambda \mid A$
$A \rightarrow aA$

A is useless variable and the production $S \rightarrow A$ plays no role since A cannot be eventually transformed into a terminal string; while A can appear in a sentential form derived from S, this sentential form can never lead to sentence!

Hence, removing $S \rightarrow A$ (and $A \rightarrow aA$) does not change the language, but does simplify the grammar.

- For a grammar with productions

$S \rightarrow A$
$A \rightarrow aA \mid \lambda$
$B \rightarrow bA$

B is useless so is the production $B \rightarrow bA$! Observe that, even though a terminal string can be derived from B, there is no way to get to B from S, i.e. cannot achieve $S \Rightarrow^* xBy$.

- **Example:**
Eliminate useless symbols and productions from $G = (V, T, S, P)$, where $V = \{S, A, B, C\}$, $T = \{a, b\}$ and P consists of

$S \rightarrow aS \mid A \mid C$
$A \rightarrow a$
$B \rightarrow aa$
$C \rightarrow aCb$

First, note that the variable C cannot lead to any terminal string, we can then remove C and its associated productions, we get G_1 with $V_1 = \{S, A, B\}$, $T_1 = \{a\}$ and P_1 consisting of

$S \rightarrow aS \mid A$
$A \rightarrow a$
$B \rightarrow aa$

Next, we identify variables that cannot be reached from the start variable. We can create a dependency graph for V_1. For a context-free grammar, a dependency graph has its vertices labeled with variables with an edge between any two vertices I and J if there is a production of the form

$I \rightarrow xJy$

Consequently, the variable B is shown to be useless and can be removed together with its associated production.

The resulting grammar $G' = (V', T', S, P')$ is with $V' = \{S, A\}$, $T' = \{a\}$ and P' consisting of

$$S \rightarrow aS \mid A$$
$$A \rightarrow a$$

1.2 Eliminations of λ-Production

- **Definition :**
  a) Any production of a context-free grammar of the form
  $$A \rightarrow \lambda$$
  is called a λ-production.

  b) Any variable A for which the derivation
  $$A \Rightarrow^* \lambda$$
  is possible is called nullable.

- If a grammar contains some λ-productions or nullable variables but does not generate the language that contains an empty string, the λ-productions can be removed!

- **Example:**
  Consider the grammar, G with productions
  $$S \rightarrow aS_1b$$
  $$S_1 \rightarrow aS_1b \mid \lambda$$
  $L(G) = \{a^n b^n \mid n \geq 1\}$ which is a λ-free language. The λ-production can be removed after adding new productions obtained by substituting λ for S_1 on the right hand side.

  We get an equivalent G' with productions
  $$S \rightarrow aS_1b \mid ab$$
  $$S_1 \rightarrow aS_1b \mid ab$$

- **Theorem:**
  Let G be any context-free grammar with $\lambda \notin L(G)$. There exists an equivalent grammar G' without λ-productions.

  Proof :
  Find the set V_N of all nullable variables of G
  1. For all productions $A \rightarrow \lambda$, put A in V_N
  2. Repeat the following step until no further variables are added to V_N:
  For all productions
  $$B \rightarrow A_1A_2 \ldots A_n$$

  where $A_1, A_2, \ldots, A_n$ are in V_N, put B in V_N.

  With the resulting V_N, P' can be constructed by looking at all productions in P of the form
  $$A \rightarrow x_1x_2 \ldots x_m,\ m \geq 1$$

where each $x_i \in V \cup T$.

For each such production of P, we put in P' the production plus all productions generated by replacing nullable variables with λ in all possible combinations. However, if all x_i are nullable, the resulting production $A \to \lambda$ is not put in P'.

- **Example:**

For the grammar G with

$S \to ABaC$
$A \to BC$
$B \to b \mid \lambda$
$C \to D \mid \lambda$
$D \to d$

the nullable variables are A, B, and C.

The equivalent grammar G' without λ-productions has P' containing

$S \to ABaC \mid BaC \mid AaC \mid ABa \mid aC \mid Ba \mid Aa \mid a$
$A \to BC \mid C \mid B$
$B \to b$
$C \to D$
$D \to d$

1.3 Eliminations of MODULE-Production

- **Definition:**
Any production of a context-free grammar of the form

$A \to B$

where $A, B \in V$ is called a MODULE-production.

- **Theorem:**
Let $G = (V, T, S, P)$ be any context-free grammar without λ-productions. There exists a context-free grammar $G' = (V', T', S, P')$ that does not have any MODULE-productions and that is equivalent to G.

Proof:
First of all, Any MODULE-production of the form $A \to A$ can be removed without any effect. We then need to consider productions of the form $A \to B$ where A and B are different variables.

Straightforward replacement of B (with $x_1 = x_2 = \lambda$) runs into a problem when we have

$A \to B$
$B \to A$

We need to find for each A, all variables B such that

$A \Rightarrow^* B$

This can be done via a dependency graph with an edge (I, J) whenever the grammar G has a MODULE-production $I \to J$; $A \Rightarrow^* B$ whenever there is a walk from A to B in the graph.

The new grammar G' is generated by first putting in P' all non-MODULE-productions of P. Then, for all A and B with $A \Rightarrow^* B$, we add to P'

$$A \to y_1 \mid y_2 \mid \ldots \mid y_n$$

where $B \to y_1 \mid y_2 \mid \ldots \mid y_n$ is the set of all rules in P' with B on the left. Not that the rules are taken from P', therefore, none of y_i can be a single variable! Consequently, no MODULE- productions are created by this step.

- **Example:**

Consider a grammar G with

$$S \to Aa \mid B$$
$$A \to a \mid bc \mid B$$
$$B \to A \mid bb$$

We have $S \Rightarrow^* A$, $S \Rightarrow^* B$, $A \Rightarrow^* B$ and $B \Rightarrow^* A$.

First, for the set of original non-MODULE-productions, we have

$$S \to Aa$$
$$A \to a \mid bc$$
$$B \to bb$$

We then add the new rules

$$S \to a \mid bc \mid bb$$
$$A \to bb$$
$$B \to a \mid bc$$

We finally obtain the equivalent grammar G' with P' consisting of

$$S \to Aa \mid a \mid bc \mid bb$$
$$A \to a \mid bc \mid bb$$
$$B \to bb \mid a \mid bc$$

Notice that B and its associate production become useless.

2 Minimization of Context Free Grammar

- **Theorem:**

Let L be a context-free language that does not contain λ. There exists a context-free grammar that generates L and that does not have any useless productions, λ-productions or MODULE-productions.

Proof:
We need to remove the undesirable productions using the following sequence of steps.

1. Remove λ-productions
2. Remove MODULE-productions
3. Remove useless productions

3. Chomsky Normal Form

- **Definition:**
A context-free grammar is in Chomsky normal form if all productions are of the form

$$A \rightarrow BC$$

or

$$A \rightarrow a$$

where A, B, $C \in V$, and $a \in T$.

Note: that the number of symbols on the right side of productions is strictly limited; not more than two symbols.

- **Example:**
The following grammar is in Chomsky normal form.

$$S \rightarrow AS \mid a$$
$$A \rightarrow SA \mid b$$

On the other hand, the grammar below is not.

$$S \rightarrow AS \mid AAS$$
$$A \rightarrow SA \mid aa$$

- **Theorem:**
Any context-free grammar $G = (V, T, S, P)$ with $\lambda \notin L(G)$ has an equivalent grammar $G' = (V', T', S, P')$ in Chomsky normal form.

Proof:
First we assume (based on previous Theorem) without loss of generality that G has no λ-productions and no MODULE-productions. Then, we show how to construct G' in two steps.

Step 1:
Construct a grammar $G_1 = (V_1, T, S, P_1)$ from G by considering all productions in P of the form

$$A \rightarrow x_1x_2\ldots x_n$$

Where each x_i is a symbol either in V or in T.

Note that if $n = 1$, x_1 must be a terminal because there is no MODULE-productions in G. In this case, put the production into P_1.

If $n \geq 2$, introduce new variables B_a for each $a \in T$. Then, for each production of the form $A \rightarrow x_1x_2\ldots x_n$, we shall remove all terminals from productions whose right side has length greater than one

This is done by putting into P_1 a production

$$A \rightarrow C_1C_2\ldots C_n$$

Where

$$C_i = x_i \text{ if } x_i \in V$$

And

$$C_i = B_a \text{ if } x_i = a$$

And, for every B_a, we also put into P_1 a production

$$B_a \rightarrow a$$

As a consequence of Theorem 6.1, it can be claimed that

$$L(G_1) = L(G)$$

Step 2:

The length of right side of productions is reduced by means of additional variables wherever necessary. First of all, all productions with a single terminal or two variables ($n = 2$) are put into P'. Then, for any production with $n > 2$, new variables D_1, D_2, … are introduced and the following productions are put into P'.

$$A \rightarrow C_1D_1$$
$$D_1 \rightarrow C_2D_2$$
$$\ldots$$
$$D_{n-2} \rightarrow C_{n-1}C_n$$

G' is clearly in Chomsky normal form.

- **Example:**

Convert to Chomsky normal form the following grammar G with productions.

$$S \rightarrow ABa$$
$$A \rightarrow aab$$
$$B \rightarrow Ac$$

Solution:

Step 1:

New variables B_a, B_b, B_c are introduced and a new grammar G_1 is obtained.

$$S \rightarrow ABB_a$$
$$A \rightarrow B_aB_aB_b$$
$$B \rightarrow AB_c$$
$$B_a \rightarrow a$$
$$B_b \rightarrow b$$
$$B_c \rightarrow c$$

Step 2:

Additional variables are introduced to reduce the length of the first two productions making them into the normal form, we finally obtain G'.

$S \rightarrow AD_1$
$D_1 \rightarrow BB_a$
$A \rightarrow B_aD_2$
$D_2 \rightarrow B_aB_b$
$B \rightarrow AB_c$
$B_a \rightarrow a$
$B_b \rightarrow b$
$B_c \rightarrow c$

Greibach normal form

- **Definition:**
  A context-free grammar is said to be in Greibach normal form if all productions have the form

  $$A \rightarrow ax$$

  where $a \in T$ and $x \in V^*$

  Note that the restriction here is not on the number of symbols on the right side, but rather on the positions of the terminals and variables.

- **Example:**
  The following grammar is not in Greibach normal form.

  $S \rightarrow AB$
  $A \rightarrow aA \mid bB \mid b$
  $B \rightarrow b$

  It can, however, be converted to the following equivalent grammar in Greibach normal form.

  $S \rightarrow aAB \mid bBB \mid bB$
  $A \rightarrow aA \mid bB \mid b$
  $B \rightarrow b$

- **Theorem:**
  For every context-free grammar G with $\lambda \notin L(G)$, there exists an equivalent grammar G' in Greibach normal form.

Conversion

- Convert from Chomsky to Greibach in two steps:
  1. From Chomsky to intermediate grammar
     a) Eliminate direct left recursion
     b) Use $A \rightarrow uBv$ rules transformations to improve references (explained later)

2. From intermediate grammar into Greibach

1.a) Eliminate direct left recursion

Step1:

- Before

  $A \rightarrow A\underline{a} \mid \boldsymbol{b}$

- After

  $A \rightarrow \boldsymbol{b}Z \mid \boldsymbol{b}$

  $Z \rightarrow \underline{a}Z \mid \underline{a}$

- Remove the rule with direct left recursion, and create a new one with recursion on the right

Step2:

- Before

  $A \rightarrow A\underline{a} \mid A\underline{b} \mid \boldsymbol{b} \mid \boldsymbol{c}$

- After

  $A \rightarrow \boldsymbol{b}\underline{Z} \mid \boldsymbol{c}\underline{Z} \mid \boldsymbol{b} \mid \boldsymbol{c}$

  $Z \rightarrow \underline{a}Z \mid \underline{b}Z \mid \underline{a} \mid \underline{b}$

- Remove the rules with direct left recursion, and create new ones with recursion on the right

Step3:

- Before

  $A \rightarrow A\underline{B} \mid \boldsymbol{BA} \mid \boldsymbol{a}$

  $B \rightarrow b \mid c$

- After

  $A \rightarrow \boldsymbol{BA}\underline{Z} \mid \boldsymbol{a}\underline{Z} \mid \boldsymbol{BA} \mid \boldsymbol{a}$

  $Z \rightarrow \underline{B}Z \mid \underline{B}$

  $B \rightarrow b \mid c$

Transform $A \rightarrow uBv$ rules

- Before

  $A \rightarrow uBb$

  $B \rightarrow w_1 \mid w_1 \mid ... \mid w_n$

- After

  Add $A \rightarrow uw_1b \mid uw_1b \mid ... \mid uw_nb$

  Delete $A \rightarrow uBb$

Background Information for the Pumping Lemma for Context-Free Languages

- **Definition:** Let G = (V, T, P, S) be a CFL. If every production in P is of the form

  $A \rightarrow BC$

  or $A \rightarrow a$

  where A, B and C are all in V and a is in T, then G is in Chomsky Normal Form (CNF).

- **Example:**

  $S \rightarrow AB \mid BA \mid aSb$
  $A \rightarrow a$
  $B \rightarrow b$

- **Theorem:** Let L be a CFL. Then L – {ε} is a CFL.

- **Theorem:** Let L be a CFL not containing {ε}. Then there exists a CNF grammar G such that L = L(G).

- **Definition:** Let T be a tree. Then the height of T, denoted h(T), is defined as follows:
  - If T consists of a single vertex then h(T) = 0
  - If T consists of a root r and subtrees $T_1, T_2, \ldots T_k$, then $h(T) = \max_i\{h(T_i)\} + 1$

- **Lemma:** Let G be a CFG in CNF. In addition, let w be a string of terminals where A=>*w and w has a derivation tree T. If T has height h(T)≥1, then $|w| \leq 2^{h(T)-1}$.

- **Proof:** By induction on h(T) (exercise).

- **Corollary:** Let G be a CFG in CNF, and let w be a string in L(G). If $|w| \geq 2^k$, where k ≥ 0, then any derivation tree for w using G has height at least k+1.

- **Proof:** Follows from the lemma.

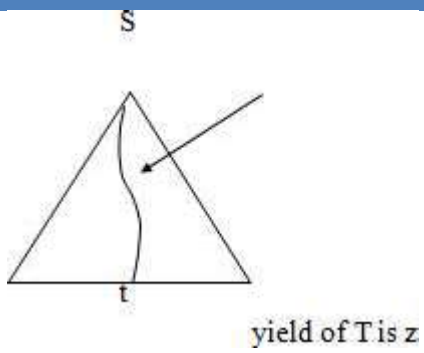
4. Pumping Lemma for Context-Free Languages

- **Lemma:**
  Let G = (V, T, P, S) be a CFG in CNF, and let $n = 2^{|V|}$. If z is a string in L(G) and |z| ≥ n, then there exist strings u, v, w, x and y in T* such that z=uvwxy and:
  - |vx| ≥ 1 (i.e., |v| + |x| ≥ 1)
  - |vwx| ≤ n
  - uv^iwx^iy is in L(G), for all i ≥ 0

- **Proof:**
  Since $|z| \geq n = 2^k$, where k = |V|, it follows from the corollary that any derivation tree for z has height at least k+1.

  By definition such a tree contains a path of length at least k+1.

  Consider the longest such path in the tree:



Such a path has:

- Length ≥ k+1 (i.e., number of edges in the path is ≥ k+1)
- At least k+2 nodes
- 1 terminal

At least k+1 non-terminals

- Since there are only k non-terminals in the grammar, and since k+1 appear on this long path, it follows that some non-terminal (and perhaps many) appears at least twice on this path.
- Consider the first non-terminal that is repeated, when traversing the path from the leaf to the root.

  This path, and the non-terminal A will be used to break up the string z.

- **In addition, (2) also tells us:**

  S =>* uAy (1)
  =>* uvAxy (2)

  =>* uv^2Ax^2y (2)
  =>* uv^2wx^2y (3)

- **More generally:**

  S =>* uv^iwx^iy for all i>=1

- **And also:**

  S =>* uAy (1)
  =>* uwy (3)

- **Hence:**

  S =>* uv^iwx^iy for all i>=0

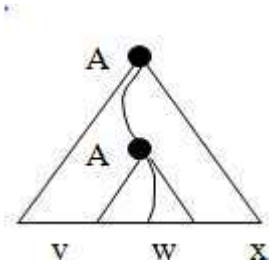
- **Consider the statement of the Pumping Lemma:**

  – *What is n?*
  n = 2^k, where k is the number of non-terminals in the grammar.

  – *Why is* $|v| + |x| \geq 1$*?*

  Since the height of this subtree is ≥ 2, the first production is A->V_1V_2. Since no non-terminal derives the empty string (in CNF), either V_1 or V_2 must derive a non-empty v or x. More specifically, if w is generated by V_1, then x contains at least one symbol, and if w is generated by V_2, then v contains at least one symbol.

  – *Why is* $|vwx| \leq n$*?*
  Observations:
  - The repeated variable was the first repeated variable on the path from the bottom, and therefore (by the pigeon-hole principle) the path from the leaf to the second occurrence of the non-terminal has length at most k+1.
  - Since the path was the largest in the entire tree, this path is the longest in the subtree rooted at the second occurrence of the non-terminal. Therefore the subtree has height ≤k+1. From the lemma, the yield of the subtree has length ≤ 2^k=n.



CFL Closure Properties

- **Theorem#1:**
  The context-free languages are closed under concatenation, union, and Kleene closure.

- **Proof:**
  Start with 2 CFL *L*(*H1*) and *L*(*H2*) generated by *H1* = (*N1*,*T1*,*R1*,*s1*) and *H2* = (*N2*,*T2*,*R2*,*s2*).
  Assume that the alphabets and rules are disjoint.

  Concatenation:
  Formed by *L*(*H1*)·*L*(*H2*) or a string in *L*(*H1*) followed by a string in *L*(*H2*) which can be

generated by $L(H3)$ generated by $H3 = (N3,T3,R3,s3)$. $N3 = N1 \cup N2$, $T3 = T1 \cup T2$, $R3 = R1 \cup R2 \cup \{s3 \text{-->} s1s2\}$ where $s3 \rightarrow s1s2$ is a new rule introduced. The new rule generates a string of L(H1) then a string of $L(H2)$. Then $L(H1) \cdot L(H2)$ is context-free.

Union:
Formed by $L(H1) \cup L(H2)$ or a string in $L(H1)$ or a string in $L(H2)$. It is generated by $L(H3)$ generated by $H4 = (N4,T4,R4,s4)$ where $N4 = N1 \cup N2$, $T4 = T1 \cup T2$, and $R4 = R1 \cup R2 \cup \{s4\text{-->}s1, s4 \rightarrow s2\}$, the new rules added will create a string of $L(H1)$ or $L(H2)$. Then $L(H1) \cup L(H2)$ is context-free.

Kleene:
Formed by $L(H1)^*$ is generated by the grammar $L(H5)$ generated by $H5 = (N1,T1,R5,s1)$ with $R5 = R1 \cup \{s1 \rightarrow e, s1 \rightarrow s1s1\}$. $L(H5)$ includes e, every string in $L(H1)$, and through i-1 applications of $s1 \rightarrow s1s1$, every string in $L(H1)i$. Then $L(H1)^*$ is generated by $H5$ and is context-free.

- **Theorem#2:**
The set of context-free languages is not closed under complementation or intersection.

- **Proof:**
Intersections of two languages $L1 \cap L2$ can be defined in terms of the Complement and Union operations as follows:
$L1 \cap L2 = \Sigma^* - (\Sigma^* - L1) \cup (\Sigma^* - L2)$

Therefore if CFL are closed under intersection then it is closed under compliment and if closed under compliment then it is closed under intersection.

The proof is just showing two context-free languages that their intersection is not a context-free language.

Choose $L1 = \{anbncm \mid m,n \geq 0\}$ is generated by grammar $H1 = \{N1,T1,R1,s1\}$, where
$N1 = \{s, A, B\}$
$T1 = \{a, b, c\}$
$R1 = \{s \rightarrow AB,$
$A \rightarrow aAb,$
$A \rightarrow e,$
$B \rightarrow Bc,$
$B \rightarrow e\}.$

Choose $L2 = \{ambncn \mid m,n$ $H2 = \{N2,T2,R2,s2\}$, where
$N1 = \{s, A, B\}$
$T1 = \{a, b, c\}$
$R2 = \{s \rightarrow AB,$
$A \rightarrow aA,$
$A \rightarrow e,$
$B \rightarrow bBc,$
$B \rightarrow e\}.$

Thus *L1* and *L2* are both context-free.

The intersection of the two languages is *L3* = {*anbncn* | *n*
already been proven earlier in this paper to be not context-free. Therefore CFL are not closed under intersections, which also means that it is not closed under complementation.

5. Pushdown Automata (PDA)

- **Informally:**

– A PDA is an NFA-ε with a stack.

–Transitions are modified to accommodate stack operations.

- **Questions:**

–What is a stack?

–How does a stack help?

- A DFA can "remember" only a finite amount of information, whereas a PDA can "remember" an infinite amount of (certain types of) information.

- **Example:**

$\{0^n1^n \mid 0=<n\}$	Is *not* regular.
$\{0^n1^n \mid 0\leq n\leq k$, for some fixed k$\}$	Is regular, for any fixed k.

- **For k=3:**

L = {ε, 01, 0011, 000111}

- In a DFA, each state remembers a finite amount of information.

- To get $\{0^n1^n \mid 0 \leq n\}$ with a DFA would require an infinite number of states using the preceding technique.

- An infinite stack solves the problem for $\{0^n1^n \mid 0 \leq n\}$ as follows:

—Read all 0's and place them on a stack

—Read all 1's and match with the corresponding 0's on the stack

- Only need two states to do this in a PDA

- Similarly for $\{0^n1^m0^{n+m} \mid n,m \geq 0\}$

Formal Definition of a PDA

- A pushdown automaton (PDA) is a seven-tuple:

$M = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$

Q	A finite set of states
Σ	A finite input alphabet
Γ	A finite stack alphabet
q_0	The initial/starting state, q_0 is in Q
z_0	A starting stack symbol, is in Γ
F	A set of final/accepting states, which is a subset of Q
δ	A transition function, where

$\delta: Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow$ finite subsets of $Q \times \Gamma^*$

- Consider the various parts of δ:

$Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow$ finite subsets of $Q \times \Gamma^*$

—Q on the LHS means that at each step in a computation, a PDA must consider its' current state.

—Γ on the LHS means that at each step in a computation, a PDA must consider the symbol on top of its' stack.

—$\Sigma \cup \{\varepsilon\}$ on the LHS means that at each step in a computation, a PDA may or may not consider the current input symbol, i.e., it may have epsilon transitions.

—"Finite subsets" on the RHS means that at each step in a computation, a PDA will have several

options.

–Q on the RHS means that each option specifies a new state.

–Γ* on the RHS means that each option specifies zero or more stack symbols that will replace the top stack symbol.

- **Two types of PDA transitions #1:**

$\delta(q, a, z) = \{(p_1,\gamma_1), (p_2,\gamma_2),\ldots, (p_m,\gamma_m)\}$

–Current state is q

–Current input symbol is a

–Symbol currently on top of the stack z

–Move to state p_i from q

–Replace z with γ_i on the stack (leftmost symbol on top)

–Move the input head to the next input symbol

- **Two types of PDA transitions #2:**

$\delta(q, \varepsilon, z) = \{(p_1,\gamma_1), (p_2,\gamma_2),\ldots, (p_m,\gamma_m)\}$

–Current state is q

–Current input symbol is not considered

–Symbol currently on top of the stack z

–Move to state p_i from q

–Replace z with γ_i on the stack (leftmost symbol on top)

–**No input symbol is read**

- **Example:** (balanced parentheses)

$M = (\{q_1\}, \{\text{“(“, “)”}\}, \{L, \#\}, \delta, q_1, \#, \emptyset)$

δ:

(1)	$\delta(q_1, (, \#) = \{(q_1, L\#)\}$
(2)	$\delta(q_1,), \#) = \emptyset$
(3)	$\delta(q_1, (, L) = \{(q_1, LL)\}$
(4)	$\delta(q_1,), L) = \{(q_1, \varepsilon)\}$
(5)	$\delta(q_1, \varepsilon, \#) = \{(q_1, \varepsilon)\}$
(6)	$\delta(q_1, \varepsilon, L) = \emptyset$

- **Goal:** (acceptance)

–Terminate in a non-null state

–Read the entire input string

–Terminate with an empty stack

•Informally, a string is accepted if there exists a computation that uses up all the input and leaves the stack empty.

•Example Computation:

Current Input	Stack	Transition	
(())	#		
())	L#	(1)	- Could have applied rule
))	LL#	(3)	(5), but it would have
)	L#	(4)	done no good
ε	#	(4)	
ε	-	(5)	

•**Example PDA #1:** For the language $\{x \mid x = wcw^r \text{ and } w \text{ in } \{0,1\}^*\}$

$M = (\{q_1, q_2\}, \{0, 1, c\}, \{R, B, G\}, \delta, q_1, R, \emptyset)$

δ:

(1)	$\delta(q_1, 0, R) = \{(q_1, BR)\}$	(9)	$\delta(q_1, 1, R) = \{(q_1, GR)\}$
(2)	$\delta(q_1, 0, B) = \{(q_1, BB)\}$	(10)	$\delta(q_1, 1, B) = \{(q_1, GB)\}$
(3)	$\delta(q_1, 0, G) = \{(q_1, BG)\}$	(11)	$\delta(q_1, 1, G) = \{(q_1, GG)\}$
(4)	$\delta(q_1, c, R) = \{(q_2, R)\}$		
(5)	$\delta(q_1, c, B) = \{(q_2, B)\}$		
(6)	$\delta(q_1, c, G) = \{(q_2, G)\}$		
(7)	$\delta(q_2, 0, B) = \{(q_2, \varepsilon)\}$	(12)	$\delta(q_2, 1, G) = \{(q_2, \varepsilon)\}$
(8)	$\delta(q_2, \varepsilon, R) = \{(q_2, \varepsilon)\}$		

• **Notes:**

–Only rule #8 is non-deterministic.

–Rule #8 is used to pop the final stack symbol off at the end of a computation.

• **Example Computation:**

(1)	$\delta(q_1, 0, R) = \{(q_1, BR)\}$	(9)	$\delta(q_1, 1, R) = \{(q_1, GR)\}$
(2)	$\delta(q_1, 0, B) = \{(q_1, BB)\}$	(10)	$\delta(q_1, 1, B) = \{(q_1, GB)\}$
(3)	$\delta(q_1, 0, G) = \{(q_1, BG)\}$	(11)	$\delta(q_1, 1, G) = \{(q_1, GG)\}$
(4)	$\delta(q_1, c, R) = \{(q_2, R)\}$		
(5)	$\delta(q_1, c, B) = \{(q_2, B)\}$		
(6)	$\delta(q_1, c, G) = \{(q_2, G)\}$		
(7)	$\delta(q_2, 0, B) = \{(q_2, \varepsilon)\}$	(12)	$\delta(q_2, 1, G) = \{(q_2, \varepsilon)\}$
(8)	$\delta(q_2, \varepsilon, R) = \{(q_2, \varepsilon)\}$		

State	Input	Stack	Rule Applied	Rules Applicable
q_1	**0**1c10	R	-	(1)
q_1	**1**c10	BR	(1)	(10)
q_1	**c**10	GBR	(10)	(6)
q_2	**1**0	GBR	(6)	(12)
q_2	**0**	BR	(12)	(7)
q_2	**ε**	R	(7)	(8)
q_2	ε	ε	(8)	-

• **Example Computation:**

(1)	$\delta(q_1, 0, R) = \{(q_1, BR)\}$	(9)	$\delta(q_1, 1, R) = \{(q_1, GR)\}$
(2)	$\delta(q_1, 0, B) = \{(q_1, BB)\}$	(10)	$\delta(q_1, 1, B) = \{(q_1, GB)\}$
(3)	$\delta(q_1, 0, G) = \{(q_1, BG)\}$	(11)	$\delta(q_1, 1, G) = \{(q_1, GG)\}$
(4)	$\delta(q_1, c, R) = \{(q_2, R)\}$		
(5)	$\delta(q_1, c, B) = \{(q_2, B)\}$		
(6)	$\delta(q_1, c, G) = \{(q_2, G)\}$		
(7)	$\delta(q_2, 0, B) = \{(q_2, \varepsilon)\}$	(12)	$\delta(q_2, 1, G) = \{(q_2, \varepsilon)\}$
(8)	$\delta(q_2, \varepsilon, R) = \{(q_2, \varepsilon)\}$		

State	Input	Stack	Rule Applied
q_1	**1**c1	R	
q_1	**c**1	GR	(9)
q_2	**1**	GR	(6)
q_2	**ε**	R	(12)
q_2	ε	ε	(8)

• **Definition:** |—* is the reflexive and transitive closure of |—.

– I |—* I for each instantaneous description I

– If I |— J and J |—* K then I |—* K

•Intuitively, if I and J are instantaneous descriptions, then I |—* J means that J follows from I by zero or more transitions.

•**Definition:** Let M = (Q, Σ, Γ, δ, q_0, z_0, F) be a PDA. The *language accepted by empty stack*, denoted $L_E(M)$, is the set

$$\{w \mid (q_0, w, z_0) \vdash^* (p, \varepsilon, \varepsilon) \text{ for some } p \text{ in } Q\}$$

•**Definition:** Let M = (Q, Σ, Γ, δ, q_0, z_0, F) be a PDA. The *language accepted by final state*, denoted $L_F(M)$, is the set

$$\{w \mid (q_0, w, z_0) \vdash^* (p, \varepsilon, \gamma) \text{ for some } p \text{ in } F \text{ and } \gamma \text{ in } \Gamma^*\}$$

•**Definition:** Let M = (Q, Σ, Γ, δ, q_0, z_0, F) be a PDA. The *language accepted by empty stack and final state*, denoted L(M), is the set

$$\{w \mid (q_0, w, z_0) \vdash^* (p, \varepsilon, \varepsilon) \text{ for some } p \text{ in } F\}$$

•**Lemma 1:** Let $L = L_E(M_1)$ for some PDA M_1. Then there exits a PDA M_2 such that $L = L_F(M_2)$.

•**Lemma 2:** Let $L = L_F(M_1)$ for some PDA M_1. Then there exits a PDA M_2 such that $L = L_E(M_2)$.

•**Theorem:** Let L be a language. Then there exits a PDA M_1 such that $L = L_F(M_1)$ if and only if there exists a PDA M_2 such that $L = L_E(M_2)$.

•**Corollary:** The PDAs that accept by empty stack and the PDAs that accept by final state define the same class of languages.

•**Note:** Similar lemmas and theorems could be stated for PDAs that accept by both final state and empty stack.

Greibach Normal Form (GNF)

•**Definition:** Let G = (V, T, P, S) be a CFL. If every production in P is of the form

$$A \rightarrow a\alpha$$

Where A is in V, a is in T, and α is in V*, then G is said to be in Greibach Normal Form (GNF).

•**Example:**

$S \rightarrow aAB \mid bB$
$A \rightarrow aA \mid a$

B –> bB | c

- **Theorem:** Let L be a CFL. Then L – {ε} is a CFL.

- **Theorem:** Let L be a CFL not containing {ε}. Then there exists a GNF grammar G such that L = L(G).

- **Lemma 1:** Let L be a CFL. Then there exists a PDA M such that L = L_E(M).

- **Proof:** Assume without loss of generality that ε is not in L. The construction can be modified to include ε later.

Let G = (V, T, P, S) be a CFG, and assume without loss of generality that G is in GNF.
Construct M = (Q, Σ, Γ, δ, q, z, Ø) where:

Q = {q}
Σ = T
Γ = V
z = S

δ: for all a in Σ and A in Γ, δ(q, a, A) contains (q, γ) if A –> aγ is in P or rather:
δ(q, a, A) = {(q, γ) | A –> aγ is in P and γ is in Γ*}, for all a in Σ and A in Γ

- For a given string x in Σ* , M will attempt to simulate a leftmost derivation of x with G.

- **Example #1:** Consider the following CFG in GNF.

S → aS
S → a

G is in GNF
L(G) = a+

Construct M as:
Q = {q}
Σ = T = {a}
Γ = V = {S}
z = S

δ(q, a, S) = {(q, S), (q, ε)}
δ(q, ε, S) = Ø

- **Example #2:** Consider the following CFG in GNF.

(1) S –> aA
(2) S –> aB
(3) A –> aA
(4) A –> aB

G is in GNF
L(G) = a+b+

(5) B –> bB
(6) B –> b

Construct M as:

Q = {q}
Σ = T = {a, b}
Γ = V = {S, A, B}
z = S

(1)δ(q, a, S) = {(q, A), (q, B)} From productions #1 and 2, S->aA, S->aB
(2)δ(q, a, A) = {(q, A), (q, B)} From productions #3 and 4, A->aA, A->aB
(3) δ(q, a, B) = Ø
(4) δ(q, b, S) = Ø
(5) δ(q, b, A) = Ø
(6)δ(q, b, B) = {(q, B), (q, ε)} From productions #5 and 6, B->bB, B->b
(7) δ(q, ε, S) = Ø
(8) δ(q, ε, A) = Ø
(9) δ(q, ε, B) = Ø Recall δ: Q x (Σ U {ε}) x Γ –> finite subsets of Q x Γ*

•For a string w in L(G) the PDA M will simulate a leftmost derivation of w.

–If w is in L(G) then $(q, w, z_0) \vdash^* (q, \varepsilon, \varepsilon)$

–If $(q, w, z_0) \vdash^* (q, \varepsilon, \varepsilon)$ then w is in L(G)

•Consider generating a string using G. Since G is in GNF, each sentential form in a *leftmost* derivation has form:

•And each step in the derivation (i.e., each application of a production) adds a terminal and some non-terminals.

$$A_1 \rightarrow t_{i+1}\alpha$$

$$\Rightarrow t_1t_2\ldots t_i\, t_{i+1}\, \alpha A_1A_2\ldots A_m$$

•Each transition of the PDA simulates one derivation step. Thus, the i^{th} step of the PDAs' computation corresponds to the i^{th} step in a corresponding leftmost derivation.

•After the i^{th} step of the computation of the PDA, $t_1t_2\ldots t_{i+1}$ are the symbols that have already

been read by the PDA and $\alpha A_1A_2\ldots A_m$ are the stack contents.

- For each leftmost derivation of a string generated by the grammar, there is an equivalent accepting computation of that string by the PDA.

- Each sentential form in the leftmost derivation corresponds to an instantaneous description in the PDA's corresponding computation.

- For example, the PDA instantaneous description corresponding to the sentential form:

$$=> t_1t_2\ldots t_i A_1A_2\ldots A_m$$

would be: $(q, t_{i+1}t_{i+2}\ldots t_n, A_1A_2\ldots A_m)$

- **Example:** Using the grammar from example #2:

S => aA	(1)
=> aaA	(3)
=> aaaA	(3)
=> aaaaB	(4)
=> aaaabB	(5)
=> aaaabb	(6)

- The corresponding computation of the PDA:

• (q, aaaabb, S)	|— (q, aaabb, A)	(1)/1
	|— (q, aabb, A)	(2)/1
	|— (q, abb, A)	(2)/1
	|— (q, bb, B)	(2)/2
	|— (q, b, B)	(6)/1
	|— (q, ε, ε)	(6)/2

–String is read
–Stack is emptied
–Therefore the string is accepted by the PDA

- **Example #3:** Consider the following CFG in GNF.

(1)	S –> aABC	
(2)	A –> a	G is in GNF
(3)	B –> b	
(4)	C –> cAB	
(5)	C –> cC	

Construct M as:

Q = {q}
Σ = T = {a, b, c}
Γ = V = {S, A, B, C}
z = S

(1)	δ(q, a, S) = {(q, ABC)}	S->aABC	(9)	δ(q, c, S) = Ø
(2)	δ(q, a, A) = {(q, ε)}	A->a	(10)	δ(q, c, A) = Ø
(3)	δ(q, a, B) = Ø		(11)	δ(q, c, B) = Ø
(4)	δ(q, a, C) = Ø	C->cAB|cC	(12)	δ(q, c, C) = {(q, AB), (q, C))
(5)	δ(q, b, S) = Ø		(13)	δ(q, ε, S) = Ø
(6)	δ(q, b, A) = Ø		(14)	δ(q, ε, A) = Ø
(7)	δ(q, b, B) = {(q, ε)}	B->b	(15)	δ(q, ε, B) = Ø
(8)	δ(q, b, C) = Ø		(16)	δ(q, ε, C) = Ø

- **Notes:**

–Recall that the grammar G was required to be in GNF before the construction could be applied.
–As a result, it was assumed at the start that ε was not in the context-free language L.

- **Suppose ε is in L:**

1) First, let L' = L – {ε}

Fact: If L is a CFL, then L' = L – {ε} is a CFL.

By an earlier theorem, there is GNF grammar G such that L' = L(G).

2) Construct a PDA M such that L' = L_E(M)

How do we modify M to accept ε?

Add δ(q, ε, S) = {(q, ε)}? No!

- **Counter Example:**

Consider L = {ε, b, ab, aab, aaab, …}
Then L' = {b, ab, aab, aaab, …}

•**The GNF CFG for L':**

(1) S –> aS
(2) S –> b

•**The PDA M Accepting L':**

Q = {q}
Σ = T = {a, b}
Γ = V = {S}
z = S

δ(q, a, S) = {(q, S)}
δ(q, b, S) = {(q, ε)}
δ(q, ε, S) = Ø

•If δ(q, ε, S) = {(q, ε)} is added then:

L(M) = {ε, a, aa, aaa, …, b, ab, aab, aaab, …}

3) Instead, add a new *start* state q' with transitions:

δ(q', ε, S) = {(q', ε), (q, S)}

•**Lemma 1:** Let L be a CFL. Then there exists a PDA M such that L = L_E(M).

•**Lemma 2:** Let M be a PDA. Then there exists a CFG grammar G such that L_E(M) = L(G).

•**Theorem:** Let L be a language. Then there exists a CFG G such that L = L(G) iff there exists a PDA M such that L = L_E(M).

•**Corollary:** The PDAs define the CFLs.

Equivalence of CFG to PDAs

- **Example:** Consider the grammar for arithmetic expressions we introduced earlier. It is reproduced below for convenience. G = ({E, T, F}, {n, v, +, *, (,)}, P, E), where

E = { 1: E → E + T,
2: E → T,
3: T → T* F,
4: T → F,
5: F → n,
6: F → v,
7: F → (E),
}

Suppose the input to our parser is the expression, n*(v+n*v). Since G is unambiguous this expression has only one leftmost derivation, p = 2345712463456. We describe the behavior of the PDA in general, and then step through its moves using this derivation to guide the computation.

- **PDA Simulator:**
  - Step 1: Initialize the stack with the start symbol (E in this case). The start symbol will serve as the bottom of stack marker (Z_0).
  - Step 2: Ignoring the input, check the top symbol of the stack.
    - Case (a) Top of stack is a nonterminal, "X": non-deterministically decide which
      X-rule to use as the next step of the derivation. After selecting a rule, replace X in the stack with the rightpart of that rule. If the stack is non-empty, repeat step 2. Otherwise, halt (input may or may not be empty.)
    - Case(b) Top of stack is a terminal, "a": Read the next input. If the input matches a, then pop the stack and repeat step 2.
      Otherwise, halt (without popping "a" from the stack.)
  - This parsing algorithm by showing the sequence of configurations the parser would assume in an accepting computation for the input, n*(v+n*v).
    Assume "q0" is the one and only state of this PDA.
  - p (leftmost derivation in G) = 2345712463456

    (q0, n*(v+n*v), E)

    2⇒M (q0, n*(v+n*v), T)

    3⇒M (q0, n*(v+n*v), T*F)

    4⇒M (q0, n*(v+n*v), F*F)

5⇒M (q0, n*(v+n*v), n*F) read⇒M (q0, *(v+n*v), *F)

read⇒M (q0, (v+n*v), F)

7⇒M (q0, (v+n*v), (E)) read⇒M (q0, v+n*v), E))

1⇒M (q0, v+n*v),E+T))

2⇒M (q0, v+n*v), T+T))

4⇒M (q0, v+n*v), F+T))

6⇒M (q0, v+n*v), v+T)) read⇒M (q0, +n*v), +T))

read⇒M (q0, n*v), T))

3⇒M (q0, n*v), T*F))

4⇒M (q0, n*v), F*F))

5⇒M (q0, n*v), n*F)) read⇒M (q0, *v), *F))

read⇒M (q0, v), F))

6⇒M (q0, v), v)) read⇒M (q0,),))

read⇒M (q0, l, l) accept!

Deterministic PDAs and DCFLs

- **Definition:** A *Deterministic Pushdown Automaton* (DPDA) is a 7-tuple,

  $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, A)$,
  where
  Q = finite set of states,
  Σ = input alphabet,
  Γ = stack alphabet,

  $q_0 \in Q$ = the initial state,
  $Z_0 \in \Gamma$ = bottom of stack marker (or initial stack symbol), and

  $\delta: Q \times (\Sigma \cup \{L\}) \times \Gamma \rightarrow Q \times \Gamma^*$ = the transition function (not necessarily total).
  Specifically,

  [1] if $d(q, a, Z)$ is defined for some $a \in \Sigma$ and $Z \in \Gamma$, then $d(q, L, Z) = \Phi$ and $|d(q, a, Z)| = 1$.

[2] Conversely, if $d(q, L, Z) \neq \Phi$, for some Z, then $d(q, a, Z) = \Phi$, for all $a \in \Sigma$, and $|d(q, L, Z)| = 1$.

- **NOTE:** DPDAs can accept their input either by final state or by empty stack – just as for the non-deterministic model. We therefore define D_{stk} and D_{ste}, respectively, as the corresponding families of Deterministic Context-free Languages accepted by a DPDA by empty stack and final state.