Module-3
Context – Free Grammars

Context Free Grammar is formal grammar, the syntax or structure of a formal language can be described using context-free grammar (CFG). A context-free grammar is a formal notation for expressing recursive definitions of languages.

Definition of Context-Free Grammars:

The context-free grammar is defined by quadruple,

G=(V, T, P, S)

where,

- V is a finite set of variables, also called sometimes non-terminals. Each variable represents a language i.e., a set of strings.
- T is a finite set of symbols that form the strings of the language being defined. We call this alphabet the terminals or terminal symbols.
- S is the start symbol, is one of the variables represents the language being defined.
- There is a finite set of productions or rules that represent the recursive definition of a language. Each production consists of:
  a) A variable that is being defined by the production. This variable is often called the head of the production.
  b) The production symbol →
  c) A string of zero or more terminals and variables. This string, called the body of the production, represents one way to form strings in the language of the variable of the head.

Example: Consider the grammar for arithmetic expression as follows:

G= (V, T, S, P) with

V={E}

T={+, *, id}

P={ **E→E+E**

E→E-E

E→E*E

E→E/E

E→ id }

and E is the start symbol.

Derivations using a Grammar:

"The process of obtaining string of terminals and/or non-terminals from the start symbol by applying some or all productions is called derivation".

The process of deriving strings by applying productions from head to body requires the definition of a new relation symbol $\Rightarrow$. Suppose $G = (V, T, P, S)$ is a CFG. Let $\alpha A \beta$ be a string of terminals and variables, with A a variable. That is, α and β are strings in $(V \cup T)^*$, and A is in V. Let $A \rightarrow \gamma$ be a production of G. Then we say $\alpha A \beta \underset{G}{\Rightarrow} \alpha \gamma \beta$. If G is understood, we just say $\alpha A \beta \Rightarrow \alpha \gamma \beta$. Notice that one derivation step replaces any variable anywhere in the string by the body of one of its productions.

We may extend the $\Rightarrow$ relationship to represent zero, one, or many derivation steps, much as the transition function δ of a finite automaton was extended to $\hat{\delta}$. For derivations, we use a * to denote "zero or more steps," as follows:

Let us derive the sentence **id+id*id** from the grammar for arithmetic expression:

E => E+E

=>id+E

=>id+E*E

=>id+id*E

=>id+id*id

Leftmost and Rightmost Derivations

LMD:

In order to restrict the number of choices we have in deriving a string, it is often useful to require that at each step we replace the leftmost variable by one of its production bodies. Such a derivation is called a leftmost derivation, and we indicate that a derivation is leftmost by using the relations $\underset{lm}{\Rightarrow}$ and $\overset{*}{\underset{lm}{\Rightarrow}}$, for one or many steps, respectively.

RMD:

Similarly, it is possible to require that at each step the rightmost variable is replaced by one of its bodies. If so, we call the derivation rightmost and use the symbols $\underset{rm}{\Rightarrow}$ and $\overset{*}{\underset{rm}{\Rightarrow}}$ to indicate one or many rightmost derivations steps, respectively.

Ex: Consider the string w=**id+id*id**. Let us derive w using LMD and RMD as follows:

LMD:

$E \underset{lm}{\Rightarrow} E+E$

$\underset{lm}{\Rightarrow}$ id+E

$\underset{lm}{\Rightarrow}$ id+E*E

$\underset{lm}{\Rightarrow}$ id+id*E

$\underset{lm}{\Rightarrow}$ id+id*id

RMD:

$E \underset{rm}{\Rightarrow} E+E$

$\underset{rm}{\Rightarrow}$ E+E*E

$\underset{rm}{\Rightarrow}$ E+E*id

$\underset{rm}{\Rightarrow}$ E+id*id

$\underset{rm}{\Rightarrow}$ id+id*id

The Language of a Grammar:

If $G = (V, T, P, S)$ is a CFG, the *language* of G, denoted $L(G)$, is the set of terminal strings that have derivations from the start symbol. That is,

$$L(G) = \{w \text{ in } T^* \mid S \overset{*}{\underset{G}{\Rightarrow}} w\}$$

If a language L is the language of some contex-free grammar, then L is said to be context-free language or CFL.

Sentential Forms:

Derivations from the start symbol produce strings that have a special role. We call these "sentential forms." That is, if $G = (V, T, P, S)$ is a CFG, then any string α in $(V \cup T)^*$ such that $S \overset{*}{\Rightarrow} \alpha$ is a *sentential form*. If $S \overset{*}{\underset{lm}{\Rightarrow}} \alpha$, then α is a *left-sentential form*, and if $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha$, then α is a *right-sentential form*. Note that the language $L(G)$ is those sentential forms that are in T^*; i.e., they consist solely of terminals.

Examples:

Design Context free grammar for the given language:

1. Construct the CFG for the language having any number of a's over the set ∑= {a}.

   Ans :

   S→aS | ε

   Therefore, G=(V,T,S,P)

   where V={S}, T={a}, P={S→aS | ε}, S is the start symbol.

2. Construct a CFG for the regular expression (0+1)*

   Ans: S→0S | 1S | ε

   Therefore, G=(V,T,S,P)

   where V={S}, T={0,1}, P={S→0S | 1S | ε}, S is the start symbol.

3. Construct a CFG for a language L = {wcw^R | where w € (a, b)*}. Also derive the string "abbcbba"

   Ans:

   S→aSa |bSa |c

   Derivations:

   S => aSa

   S => abSba //S→bSb

   S =>abbSbba //S→bSb

   S =>abbcbba //S→c

4. Construct a CFG for the language L = $a^n b^{2n}$ where n>=1.

Ans:

S→aSbb |ab

5. L = { $a^n b^n$ | n is a positive integer }

   Ans:

   S -> aSb | ab

   therefore G=(V, T, P, S) where,

   V = { S } , T = { a , b }, P = { S -> aSb , S -> ab }and S is the start symbol.

6. Write CFG for balanced paranthesis.

   Ans:

   S→(S) | [S] | {S} |SS| ε

7. L={ww^R : w€{a,b}*}

   Ans: S→aSa |bSb | ε

8. The CFG for Plaindrome of 0's and 1's is

   Ans:

   S→aSa |bSb|a |b| ε

9. L={$0^i 1^j$ | i≠j , i>=0, j>=0}

   Ans:

   S→0S1 | A | B

   A→0A |0

   B→1B | 1

10. L={ $a^n b^m c^n$ | n>=0, m>=0}

    Ans:

    S→aAc | ε

    A→aAc | B

    B→ bB | ε

11. L={ w€{a,b}*: $n_a(w)=n_b(w)$}

    Ans: S→aSb | bSa | SS | ε

12. L={ w€{a,b}*: $n_a(w)>n_b(w)$}

    Ans: S→aSb | bSa | SS | a| aS

13. L={ w€{a,b}*: $n_a(w)<n_b(w)$}

    Ans: S→aSb | bSa | SS | b| bS

Note: Refer notes for More problems and its solutions.

Parse Trees

There is a tree representation for derivations that has proved extremely useful. This tree shows us clearly how the symbols of a terminal string are grouped into substrings, each of which belongs to the language of one of the variables of the grammar.

Constructing Parse Trees

Let us consider a grammar G=(V, T, P, S). The parse trees for G are trees with the following conditions:

1. Root of the tree is labelled by start symbol S of the grammar
2. Each interior node is labeled by a variable in V.
3. Each leaf is labeled by either a variable, a terminal, or ε. However, if the leaf is labelled ε, then it must be the only child of its parent.
4. If an interior node is labeled A, and its children are labeled

   $$X_1, X_2, \ldots\ldots\ldots\ldots, X_k$$

   respectively, from the left, then $A \rightarrow X_1 X_2 \ldots\ldots X_k$ is a production in P. Note that the only time one of the X's can be ε is if that is the label of the only child, and $A \rightarrow \varepsilon$ is a production of G.

Example: Consider the following grammar,

$S \rightarrow ASB \mid c$

$A \rightarrow \varepsilon \mid aA$

$B \rightarrow \varepsilon \mid bB$

A derivation sequence of **acb,**

S=>ASB

=>aASB

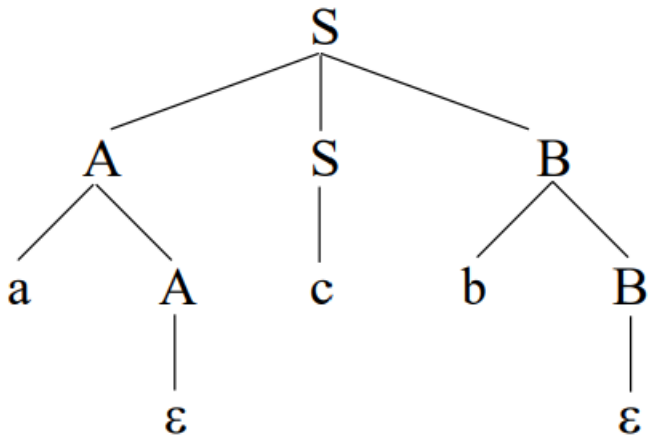
=>aSB

=>acB

=>acbB

=>acb

The Parse tree of **acb** is,



The Yield of a Parse Tree

If we look at the leaves of any parse tree and concatenate them from the left,we get a string, called the **yield** of the tree, which is always a string that is derived from the root variable. Of special importance are those parse trees such that:

1. The yield is a terminal string. That is, all leaves are labeled either with a terminal or with ε.
2. The root is labeled by the start symbol.

These are the parse trees whose yields are strings in the language of the underlying grammar.

Ex: in the above parse tree yield is, a ε c b ε=acb.

Exercise on parse tree:

1. Consider the grammar S→ (L) | a

L→L, S | S

i. What are the terminals, nonterminal and the start symbol?

ii. Find the parse tree for the following sentence

a. (a,a)

b. (a, (a, a))

c. (a, ((a,a),(a,a)))

iii. Construct LMD and RMD for each.

Ambiguity in Grammars and Languages:

Applications of CFG's often rely on the grammar to provide the structure of files. When a grammar fails to provide unique structures, it is sometimes possible to redesign the grammar to make the structure unique for each string in the language. Unfortunately, sometimes we cannot do so. That is, there are some CFL's that are "inherently ambiguous"; every grammar for the language puts more than one structure on some strings in the language.

Ambiguous Grammars:

Q. What do you mean by ambiguous grammar? Explain with example.

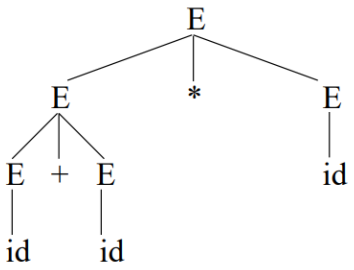
A CFG is ambiguous if it produces more than one parse tree for a string in the language. – i.e. there exists any string in the language that is the yield of two or more parse trees. In other way, if we are able to derive any string of a grammar either by applying LMD two or more time or applying RMD two or more time we call such grammar as Ambiguous grammar.

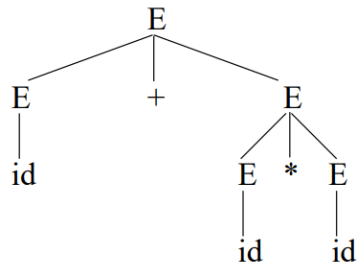
Example:

1. A good example for ambiguous grammar is the grammar for **arithmetic expressions:**

$$\mathbf{E \rightarrow E+E \mid E*E \mid E-E \mid E/E \mid id \mid (E)}$$

For the expression id+id*id we can generate 2 parse trees and 2 leftmost or 2 rightmost derivations as follows:





$E \Rightarrow_{lm} E*E \Rightarrow_{lm} E+E*E \Rightarrow_{lm} id+E*E$
$\Rightarrow_{lm} id+id*E \Rightarrow_{lm} id+id*id$

$E \Rightarrow_{lm} E+E \Rightarrow_{lm} id+E \Rightarrow_{lm} id+E*E$
$\Rightarrow_{lm} id+id*E \Rightarrow_{lm} id+id*id$

$E \Rightarrow_{rm} E*E \Rightarrow_{rm} E*id \Rightarrow_{rm} E+E*id$
$\Rightarrow_{rm} E+id*id \Rightarrow_{rm} id+id*id$

$E \Rightarrow_{rm} E+E \Rightarrow_{rm} E+E*E \Rightarrow_{rm} E+E*id$
$\Rightarrow_{rm} E+id*id \Rightarrow_{rm} id+id*id$

Since we got 2 different parse trees and LMD's we say that given grammar is ambiguous.

Removing Ambiguity From Grammars:

For the most parsers, the grammar must be unambiguous. Because,

- unambiguous grammar means unique selection of the parse tree for a sentence
- We should eliminate the ambiguity in the grammar during the design phase of the compiler.
- An ambiguous grammar should be written to eliminate the ambiguity.
- We have to prefer one of the parse trees of a sentence (generated by an ambiguous grammar) to disambiguate that grammar to restrict to this choice.

Sometimes an ambiguous grammar can be rewritten to eliminate the ambiguity.

Example 1:

There are two causes of ambiguity in the grammar of arithmetic expression,

E→ E+E | E-E |E*E |E/E |(E) | id

1. The precedence of operators is not respected.
2. A sequence of identical operators can group either from the left or from the right.

Ambiguous grammars (because of ambiguous operators) can be disambiguated according to the **precedence** and **associativity rules**. We can eliminate ambiguity of this grammar using the following precedence and associativity rules.

Precedence: from highest to lowest

*, / (left to right)

+, - (left to right)

Therefore the unambiguous grammar for arithmetic expression is,

E→E+T | E – T

T→ T*F |T/F

F→ (E) |id

Exercise Questions: Prove that the following grammars are ambiguous.

a) S→ AB | aaB

A→a | Aa

B→ b

b) S→ABA

A→aA |ε

B→bB | ε

Inherent Ambiguity

A context free language L is said to be inherently ambiguous if all its grammars are ambiguous. If even one grammar for L is unambiguous, then L is an unambiguous language.

Consider the Language,

$$L = \{a^n b^n c^m d^m \mid n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}$$

That is, L consists of strings in $\mathbf{a^+b^+c^+d^+}$ such that either:

1. There are as many a's as b's and as many c's as d's, or
2. There are as many a's as d's and as many b's as c's.

The grammar for this language is,

$$
\begin{array}{lcl}
S & \rightarrow & AB \mid C \\
A & \rightarrow & aAb \mid ab \\
B & \rightarrow & cBd \mid cd \\
C & \rightarrow & aCd \mid aDd \\
D & \rightarrow & bDc \mid bc
\end{array}
$$

Since it is impossible to derived more than one parse tree for any strings of this grammar, it is an example for inherently ambiguous grammar.

PUSHDOWN AUTOMATA

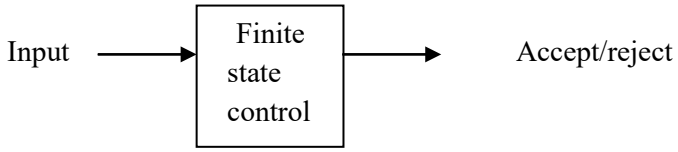
The context free languages have a type of automaton that defined them. This automaton, called a "pushdown automaton", is an extension of the nondeterministic finite automaton with ε – transitions, which is one of the ways to define the regular languages.

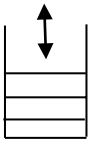
The pushdown automaton is essentially an ε – NFA with the addition of a stack. The stack canbe read, pushed and popped only at the top, just like the "stack" data structure.

We define two different versions of the pushdown automaton: one that accepts by entering an accepting state, like finite automata do and another version that accepts by emptying its stack, regardless of the state it is in. we show that these two variations accept exactly the context free languages i.e. grammars can be converted to equivalent pushdown automata and vice-versa.

Model of Pushdown Automata

- The pushdown automaton is in essence a nondeterministic finite automaton with ε – transitions permitted and one additional capability: a stack on which it can store a string of "stack symbols".
- The presence of a stack means that unlike finite automaton, the pushdown automaton can remember an infinite amount of information.
- The model of PDA is as shown below:





Stack

Figure 6.1: A Pushdown automata with stack

- As shown in the Figure 6.1, PDA has finite state control that reads inputs, one symbol at a time. The pushdown automaton is allowed to observe the symbol at the top of the stack and to base its transition on its current state, input symbol, and the symbol at the top of stack.

FORMAL DEFINITION OF PUSHDOWN AUTOMATA

- Formal definition for *pushdown automata* (PDA) involves seven components. A PDA P is defined as follows:

$$\mathbf{P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)}$$

- Where,
  - Q: a finite set of *states*, like the states of a finite automata
  - Σ: a finite set of *input symbols*, also analogous to the corresponding component of afinite automaton.
  - Γ: a *finite stack alphabet*. It is the set of symbols that we are allowed to push onto thestack.
  - δ: the *transition function*. As for a finite automaton, δ governs the behavior of the automaton.

    δ: QX Σ U {ϵ} X Γ → Q X Γ*

    Formally, δ takes as argument a triple δ(q, a, X), where:
    1. q is the state in Q
    2. a is either an input symbol in Σ or a = ε, the empty string, which is assumednot to be an input symbol.
    3. X is a stack symbol, which is a member of Γ.

    The output of δ is a finite set of pairs (p, γ), where p is the new state

and γ is the string of stack symbols that replaces X at the top of stack.

- q_0: the *start state*. The PDA is in this state before making any transitions.
- Z_0: the *start symbol*. Initially, the PDA's stack consists of one instance of this symbol,and nothing else.
- F: the set of *accepting states*, or *final states*.

A Graphical Notation for PDA's:

The transition function δ will explain the behavior of a PDA. It has following components:

a) The nodes correspond to the states of the PDA.

b) An arrow labeled *Start* indicates the start state, and doubly circled states are accepting, as for finite automata.

c) The arcs correspond to transitions of the PDA in the following sense. An arc labeled $a, X/\alpha$ from state q to state p means that $\delta(q, a, X)$ contains the pair (p, α), perhaps among other pairs. That is, the arc label tells what input is used, and also gives the old and new tops of the stack.

Example: Let us design a PDA P to accept the language $L=\{ww^R : w \in \{0,1\}^*\}$

We shall use a stack symbol Z_0 to mark the bottom of the stack. We need to have this symbol present so that, after we pop w off the stack and realize that we have seen ww^R on the input, we still have something on the stack to permit us to make a transition to the accepting state, q_2. Thus, our PDA for Lww^r can be described as:

$$P = (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, Z_0\}, \delta, q_0, Z_0, \{q_2\})$$

where δ is defined by the following rules:

1. $\delta(q_0, 0, Z_0) = \{(q_0, 0Z_0)\}$ and $\delta(q_0, 1, Z_0) = \{(q_0, 1Z_0)\}$. One of these rules applies initially, when we are in state q_0 and we see the start symbol Z_0 at the top of the stack. We read the first input, and push it onto the stack, leaving Z_0 below to mark the bottom.

2. $\delta(q_0, 0, 0) = \{(q_0, 00)\}$, $\delta(q_0, 0, 1) = \{(q_0, 01)\}$, $\delta(q_0, 1, 0) = \{(q_0, 10)\}$, and $\delta(q_0, 1, 1) = \{(q_0, 11)\}$. These four, similar rules allow us to stay in state q_0 and read inputs, pushing each onto the top of the stack and leaving the previous top stack symbol alone.

3. $\delta(q_0, \epsilon, Z_0) = \{(q_1, Z_0)\}$, $\delta(q_0, \epsilon, 0) = \{(q_1, 0)\}$, and $\delta(q_0, \epsilon, 1) = \{(q_1, 1)\}$. These three rules allow P to go from state q_0 to state q_1 spontaneously (on ϵ input), leaving intact whatever symbol is at the top of the stack.

4. $\delta(q_1, 0, 0) = \{(q_1, \epsilon)\}$, and $\delta(q_1, 1, 1) = \{(q_1, \epsilon)\}$. Now, in state q_1 we can match input symbols against the top symbols on the stack, and pop when the symbols match.

5. $\delta(q_1, \epsilon, Z_0) = \{(q_2, Z_0)\}$. Finally, if we expose the bottom-of-stack marker Z_0 and we are in state q_1, then we have found an input of the form ww^R. We go to state q_2 and accept.

Transition diagram is,

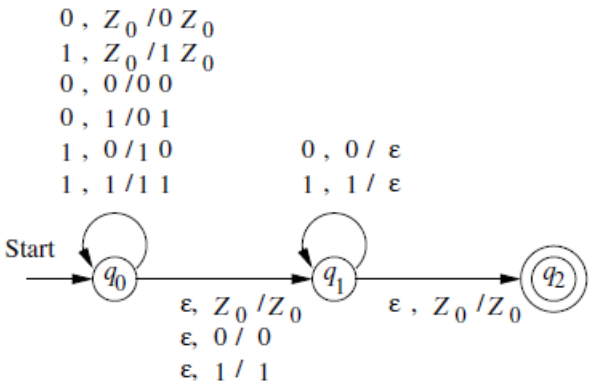


Figure 2: PDA for L={ww^R : w €{0,1}*}

Instantaneous Descriptions of a PDA:

The PDA goes from configuration to configuration, in response to input symbols (or sometimes ε) but unlike the finite automaton, where the state is the only thing that we need to know about the automaton, the PDA's configuration involves both the state and the contents of the stack. We shall represent the configuration of a PDA by a triple (q,w,γ), where:

1. **q is the state,**
2. **w is the remaining input, and**
3. **γ is the stack contents**

Such a triple is called an **instantaneous description**, or ID of the pushdown automaton.

Example:

Let us consider the action of the PDA of Example L={ww^R } on the input 1111. Since q0 is the start state and Z0 is the start symbol, the initial ID is (q0,1111,Z0). The IDs of the PDA are:

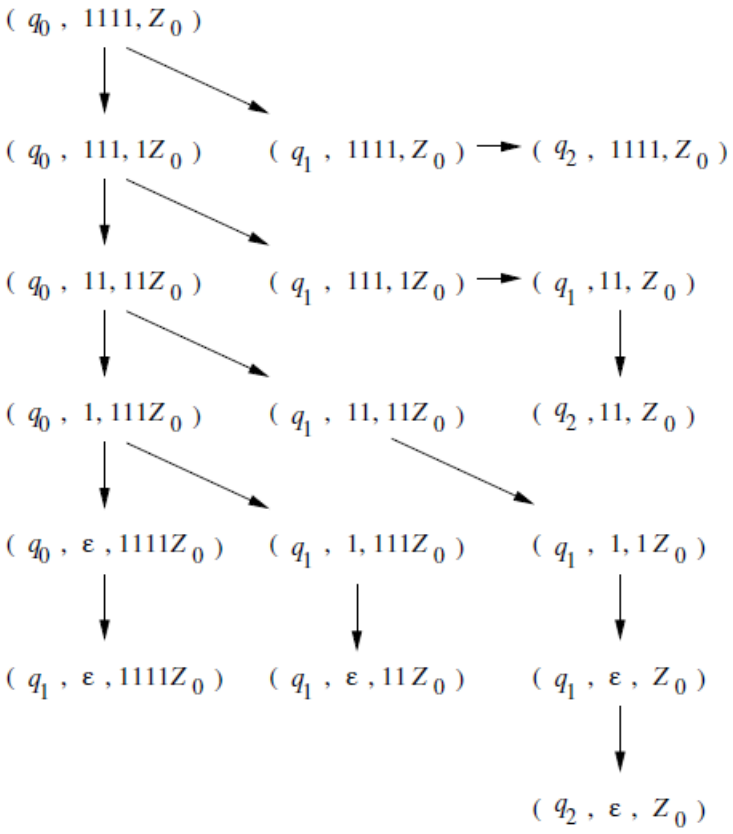


Figure 6.3: ID's of the PDA of Example 6.2 on input 1111

The Languages of a PDA:

- We have assumed that a PDA accepts it input by consuming it and entering an

accepting state. We call this approach *"acceptance by final state"*.

- There is a second approach to define the language of PDA that has important applications. We may also define for any PDA the language *"accepted by empty stack"*, i.e. the set of strings that cause the PDA to empty the stack, starting from the initial ID.
- These two methods are equivalent, in the sense that a language L has a PDA that accepts itby final state if and only if L has a PDA that accepts it by empty stack.

6.2.1 Acceptance by Final State

Let $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a PDA. Then $L(P)$, the *language accepted by* P *by final state*, is

$$\{w \mid (q_0, w, Z_0) \overset{*}{\underset{P}{\vdash}} (q, \epsilon, \alpha)\}$$

for some state q in F and any stack string α. That is, starting in the initial ID with w waiting on the input, P consumes w from the input and enters an accepting state. The contents of the stack at that time is irrelevant.

6.2.2 Acceptance by Empty Stack

For each PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, we also define

$$N(P) = \{w \mid (q_0, w, Z_0) \overset{*}{\vdash} (q, \epsilon, \epsilon)\}$$

for any state q. That is, $N(P)$ is the set of inputs w that P can consume and at the same time empty its stack.[2]

EQUIVALENCE OF ACCEPTANCE BY FINAL STATE AND EMPTY STACK

From Empty Stack to Final State:

Theorem 6.9 : If $L = N(P_N)$ for some PDA $P_N = (Q, \Sigma, \Gamma, \delta_N, q_0, Z_0)$, then there is a PDA P_F such that $L = L(P_F)$.

PROOF: The idea behind the proof is in Fig. 6.4. We use a new symbol X_0, which must not be a symbol of Γ; X_0 is both the start symbol of P_F and a marker on the bottom of the stack that lets us know when P_N has reached an empty stack. That is, if P_F sees X_0 on top of its stack, then it knows that P_N would empty its stack on the same input.

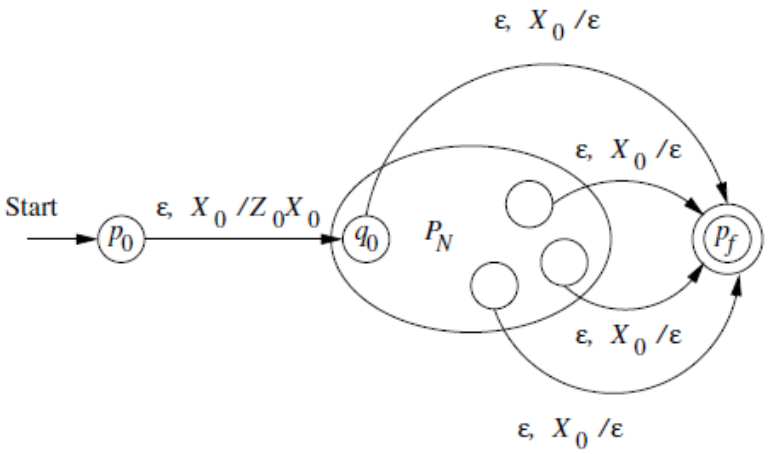


Figure 6.4: P_F simulates P_N and accepts if P_N empties its stack

We also need a new start state, p_0, whose sole function is to push Z_0, the start symbol of P_N, onto the top of the stack and enter state q_0, the start state of P_N. Then, P_F simulates P_N, until the stack of P_N is empty, which P_F detects because it sees X_0 on the top of the stack. Finally, we need another new state, p_f, which is the accepting state of P_F; this PDA transfers to state p_f whenever it discovers that P_N would have emptied its stack.

The specification of P_F is as follows:

$$P_F = (Q \cup \{p_0, p_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta_F, p_0, X_0, \{p_f\})$$

where δ_F is defined by:

1. $\delta_F(p_0, \epsilon, X_0) = \{(q_0, Z_0X_0)\}$. In its start state, P_F makes a spontaneous transition to the start state of P_N, pushing its start symbol Z_0 onto the stack.

2. For all states q in Q, inputs a in Σ or $a = \epsilon$, and stack symbols Y in Γ, $\delta_F(q, a, Y)$ contains all the pairs in $\delta_N(q, a, Y)$.

3. In addition to rule (2), $\delta_F(q, \epsilon, X_0)$ contains (p_f, ϵ) for every state q in Q.

We must show that w is in $L(P_F)$ if and only if w is in $N(P_N)$.

(If) We are given that $(q_0, w, Z_0) \overset{*}{\underset{P_N}{\vdash}} (q, \epsilon, \epsilon)$ for some state q. Theorem 6.5 lets us insert X_0 at the bottom of the stack and conclude $(q_0, w, Z_0X_0) \overset{*}{\underset{P_N}{\vdash}} (q, \epsilon, X_0)$. Since by rule (2) above, P_F has all the moves of P_N, we may also conclude that $(q_0, w, Z_0X_0) \overset{*}{\underset{P_F}{\vdash}} (q, \epsilon, X_0)$. If we put this sequence of moves together with the initial and final moves from rules (1) and (3) above, we get:

$$(p_0, w, X_0) \underset{P_F}{\vdash} (q_0, w, Z_0X_0) \overset{*}{\underset{P_F}{\vdash}} (q, \epsilon, X_0) \underset{P_F}{\vdash} (p_f, \epsilon, \epsilon) \tag{6.1}$$

Thus, P_F accepts w by final state.

From Final State to Empty Stack

Now, let us go in the opposite direction: take a PDA P_F that accepts a language L by final state and construct another PDA P_N that accepts L by empty stack. The construction is simple and is suggested in Fig. 6.7. From each accepting state of P_F, add a transition on ϵ to a new state p. When in state p, P_N pops its stack and does not consume any input. Thus, whenever P_F enters an accepting state after consuming input w, P_N will empty its stack after consuming w.

To avoid simulating a situation where P_F accidentally empties its stack without accepting, P_N must also use a marker X_0 on the bottom of its stack. The marker is P_N's start symbol, and like the construction of Theorem 6.9, P_N must start in a new state p_0, whose sole function is to push the start symbol of P_F on the stack and go to the start state of P_F. The construction is sketched in Fig. 6.7, and we give it formally in the next theorem.

Theorem 6.11: Let L be $L(P_F)$ for some PDA $P_F = (Q, \Sigma, \Gamma, \delta_F, q_0, Z_0, F)$. Then there is a PDA P_N such that $L = N(P_N)$.

PROOF: The construction is as suggested in Fig. 6.7. Let

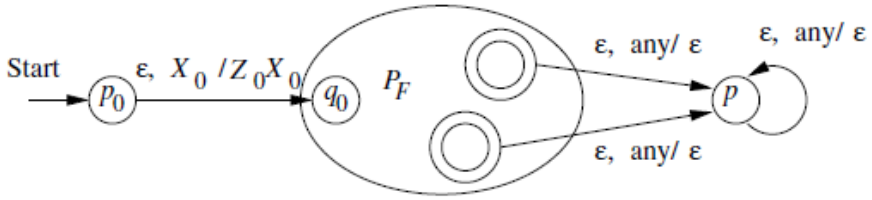


Figure 6.7: P_N simulates P_F and empties its stack when and only when P_N enters an accepting state

$$P_N = (Q \cup \{p_0, p\}, \Sigma, \Gamma \cup \{X_0\}, \delta_N, p_0, X_0)$$

where δ_N is defined by:

1. $\delta_N(p_0, \epsilon, X_0) = \{(q_0, Z_0X_0)\}$. We start by pushing the start symbol of P_F onto the stack and going to the start state of P_F.

2. For all states q in Q, input symbols a in Σ or $a = \epsilon$, and Y in Γ, $\delta_N(q, a, Y)$ contains every pair that is in $\delta_F(q, a, Y)$. That is, P_N simulates P_F.

3. For all accepting states q in F and stack symbols Y in Γ or $Y = X_0$, $\delta_N(q, \epsilon, Y)$ contains (p, ϵ). By this rule, whenever P_F accepts, P_N can start emptying its stack without consuming any more input.

4. For all stack symbols Y in Γ or $Y = X_0$, $\delta_N(p, \epsilon, Y) = \{(p, \epsilon)\}$. Once in state p, which only occurs when P_F has accepted, P_N pops every symbol on its stack, until the stack is empty. No further input is consumed.

Deterministic Pushdown Automata

Definition:

Intuitively, a PDA is deterministic if there is never a choice of move in any situation. These choices are of two kinds. If $\delta(q, a, X)$ contains more than one pair, then surely the PDA is nondeterministic because we can choose among these pairs when deciding on the next move. However, even if $\delta(q, a, X)$ is always a singleton, we could still have a choice between using a real input symbol, or making a move on ϵ. Thus, we define a PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ to be *deterministic* (a deterministic PDA or DPDA), if and only if the following conditions are met:

1. $\delta(q, a, X)$ has at most one member for any q in Q, a in Σ or $a = \epsilon$, and X in Γ.

2. If $\delta(q, a, X)$ is nonempty, for some a in Σ, then $\delta(q, \epsilon, X)$ must be empty.

Example for Deterministic PDA is,

The PDA to accept L={wCw^R : w€{0,1}*}

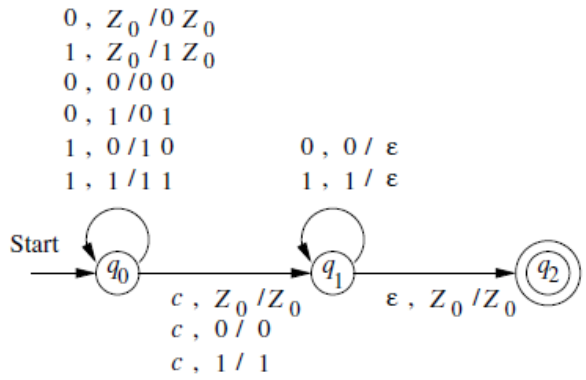
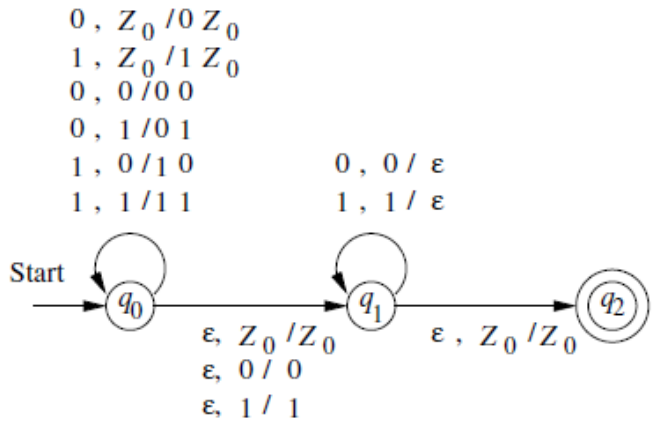


Figure 6.11: A deterministic PDA accepting L_{wcwr}

Example for Non-deterministic PDA is,

The PDA to accept L={ww^R : w€{a,b}*}



Regular Languages and Deterministic PDA's:

Theorem 6.17 : If L is a regular language, then $L = L(P)$ for some DPDA P.

PROOF: Essentially, a DPDA can simulate a deterministic finite automaton. The PDA keeps some stack symbol Z_0 on its stack, because a PDA has to have

a stack, but really the PDA ignores its stack and just uses its state. Formally, let $A = (Q, \Sigma, \delta_A, q_0, F)$ be a DFA. Construct DPDA

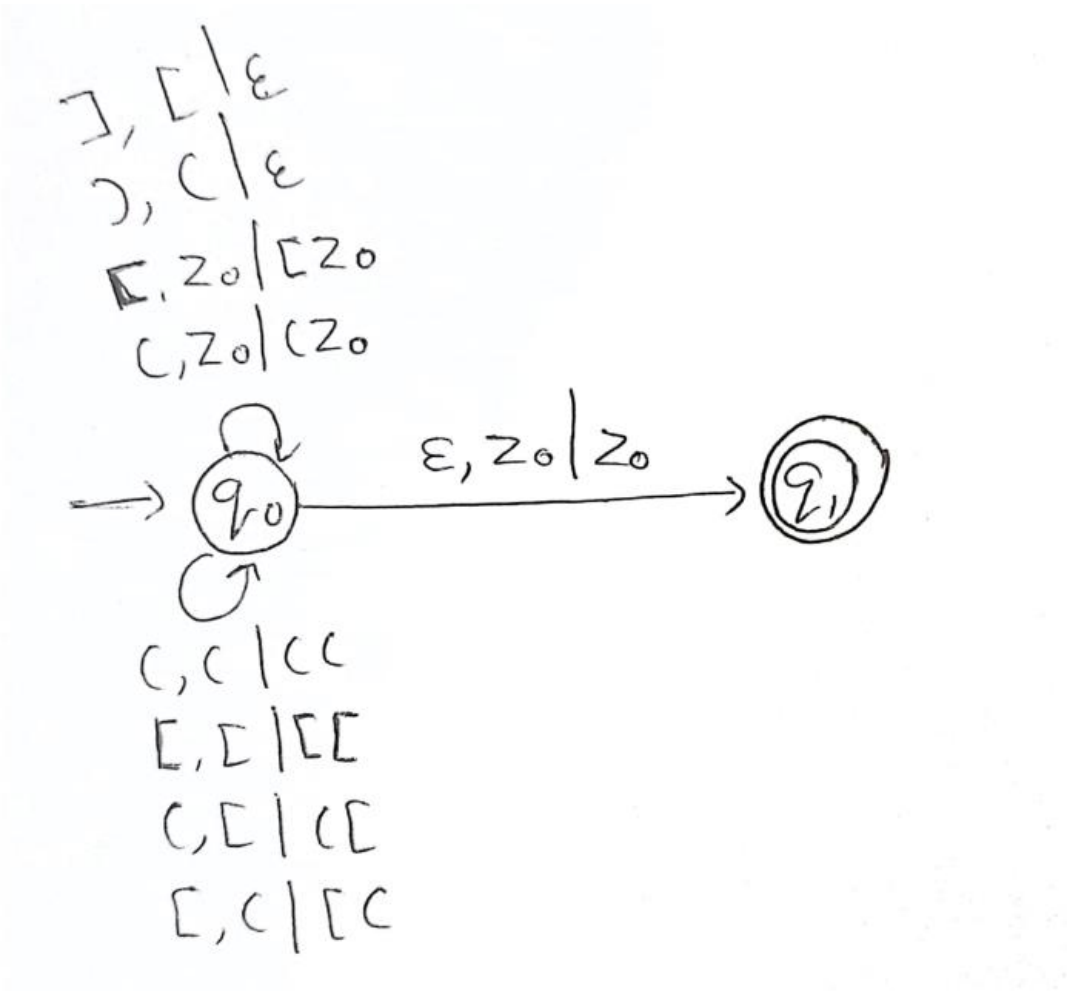
$$P = (Q, \Sigma, \{Z_0\}, \delta_P, q_0, Z_0, F)$$

by defining $\delta_P(q, a, Z_0) = \{(p, Z_0)\}$ for all states p and q in Q, such that $\delta_A(q, a) = p$.

We claim that $(q_0, w, Z_0) \overset{*}{\underset{P}{\vdash}} (p, \epsilon, Z_0)$ if and only if $\hat{\delta}_A(q_0, w) = p$. That is, P simulates A using its state. The proofs in both directions are easy inductions on $|w|$, and we leave them for the reader to complete. Since both A and P accept by entering one of the states of F, we conclude that their languages are the same. □

2. Obtain a PDA to accept a string of balanced parenthesis. The parathesis to be considered are (,), [,]. Ex: (( )), [( )( )],( ) ( )

Ans: The transition diagram for the given language is,



Therefore PDA P=(Q, ∑, Γ, δ, q_0, Z_0, F)

Where,

Q={q_0, q_1}

∑={a,b}

Γ={(, [, Z_o}

δ is as shown in the transition diagram.

q_0 is the initial state

F={q_1}

3. Construct a Pushdown automata to accept the language L={$a^n b^n$:n>=1}. Also show the IDs for the string w=aaabbb.

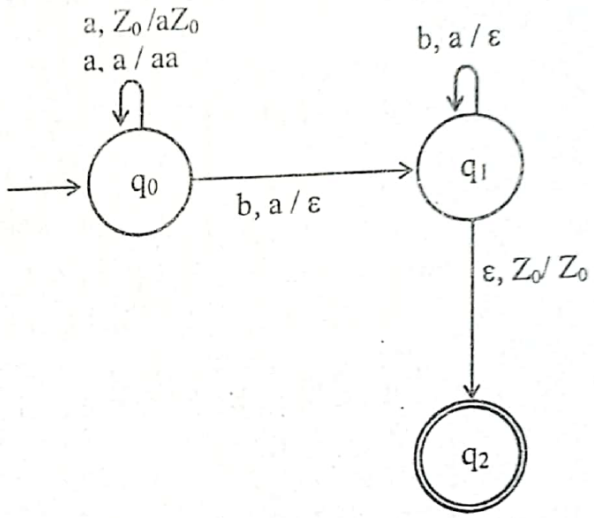
along with transition diagram is given by

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

where

$Q = \{q_0, q_1, q_2\}$
$\Sigma = \{a, b\}$
$\Gamma = \{a, Z_0\}$
δ : is shown below.

$$\delta(q_0, a, Z_0) = (q_0, aZ_0)$$
$$\delta(q_0, a, a) = (q_0, aa)$$
$$\delta(q_0, b, a) = (q_1, \varepsilon)$$
$$\delta(q_1, b, a) = (q_1, \varepsilon)$$
$$\delta(q_1, \varepsilon, Z_0) = (q_2, Z_0)$$



$q_0 \in Q$ is the start state of machine.
$Z_0 \in \Gamma$ is the initial symbol on the stack.
F = {q_2} is the final state.

Given w=aaabbb. The IDs are:

Initial ID
$(q_0, aaabbb, Z_0)$ $\vdash$ $(q_0, aabbb, aZ_0)$
$\vdash$ $(q_0, abbb, aaZ_0)$
$\vdash$ $(q_0, bbb, aaaZ_0)$
$\vdash$ (q_1, bb, aaZ_0)
$\vdash$ (q_1, b, aZ_0)
$\vdash$ (q_1, ε, Z_0)
$\vdash$ (q_2, ε, Z_0)
(Final Configuration)

Since q2 is the final state and input string is ε in the final configuration, the string aaabbb

is accepted by the PDA.

4. Construct a PDA to accept the language L={ w: w€ {a,b}* and $n_a(w)=n_b(w)$}.

$$L = \{w \mid n_a(w) = n_b(w)\}$$

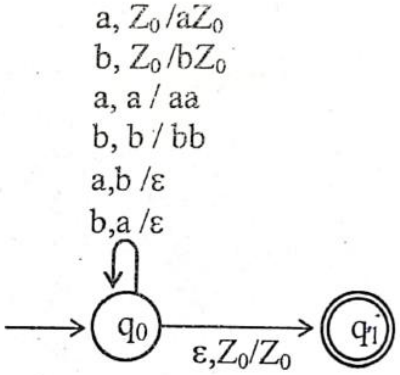
is given by

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

where

$Q = \{q_0, q_1\}$
$\Sigma = \{a, b\}$
$\Gamma = \{a, b, Z_0\}$
δ : is shown below.

$\delta(q_0, a, Z_0) = (q_0, aZ_0)$
$\delta(q_0, b, Z_0) = (q_0, bZ_0)$
$\delta(q_0, a, a) = (q_0, aa)$
$\delta(q_0, b, b) = (q_0, bb)$
$\delta(q_0, a, b) = (q_0, \varepsilon)$
$\delta(q_0, b, a) = (q_0, \varepsilon)$
$\delta(q_0, \varepsilon, Z_0) = (q_1, Z_0)$



$q_0 \in Q$ is the start state of machine.
$Z_0 \in \Gamma$ is the initial symbol on the stack.
$F = \{q_1\}$ is the final state.

Let us trace the string w=abbbaa as follows:

To accept the string: The sequence of moves made by the PDA for the string abbbaa is shown below.

Initial ID
$(q_0, abbbaa, Z_0)$ $\vdash (q_0, bbbaa, aZ_0)$
$\vdash (q_0, bbaa, Z_0)$
$\vdash (q_0, baa, bZ_0)$
$\vdash (q_0, aa, bbZ_0)$
$\vdash (q_0, a, bZ_0)$
$\vdash (q_0, \varepsilon, Z_0)$
$\vdash (q_1, \varepsilon, Z_0)$
(Final Configuration)

Since q_1 is the final state and input string is ε in the final configuration, the string

abbbaa

Note: For more problems refer class work

6.3 Equivalence of PDA's and CFG's

Now, we shall demonstrate that the languages defined by PDA's are exactly the context-free languages. The plan of attack is suggested by Fig. 6.8. The goal is to prove that the following three classes of languages:

1. The context-free languages, i.e., the languages defined by CFG's.
2. The languages that are accepted by final state by some PDA.
3. The languages that are accepted by empty stack by some PDA.

are all the same class. We have already shown that (2) and (3) are the same. It turns out to be easiest next to show that (1) and (3) are the same, thus implying the equivalence of all three.

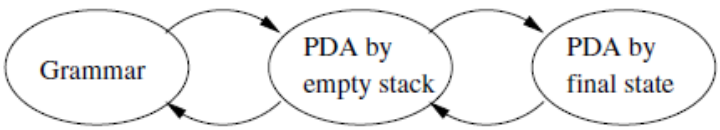


Figure 6.8: Organization of constructions showing equivalence of three ways of defining the CFL's

a)From Grammars to Pushdown Automata- By empty stack method:

(Suitable for grammars not in GNF):

Let G =(V, T, P, S) be a CFG. Construct the PDA P that accepts L(G) by empty stack as follows:

$$P = (\{q\}, T, V \cup T, \delta, q, S)$$

where transition function δ is defined by:

1. For each variable A,

$$\delta(q, \epsilon, A) = \{(q, \beta) \mid A \to \beta \text{ is a production of } G\}$$

2. For each terminal a, $\delta(q, a, a) = \{(q, \epsilon)\}$.

Example 6.12 : Let us convert the expression grammar of Fig. 5.2 to a PDA. Recall this grammar is:

$$\begin{array}{lcl} I & \to & a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\ E & \to & I \mid E * E \mid E + E \mid (E) \end{array}$$

The set of input symbols for the PDA is $\{a, b, 0, 1, (,), +, *\}$. These eight symbols and the symbols I and E form the stack alphabet. The transition function for the PDA is:

a) $\delta(q, \epsilon, I) = \{(q, a),\ (q, b),\ (q, Ia),\ (q, Ib),\ (q, I0),\ (q, I1)\}$.

b) $\delta(q, \epsilon, E) = \{(q, I),\ (q, E + E),\ (q, E * E),\ (q, (E))\}$.

c) $\delta(q, a, a) = \{(q, \epsilon)\}$; $\delta(q, b, b) = \{(q, \epsilon)\}$; $\delta(q, 0, 0) = \{(q, \epsilon)\}$; $\delta(q, 1, 1) = \{(q, \epsilon)\}$; $\delta(q, (, () = \{(q, \epsilon)\}$; $\delta(q,),)) = \{(q, \epsilon)\}$; $\delta(q, +, +) = \{(q, \epsilon)\}$; $\delta(q, *, *) = \{(q, \epsilon)\}$.

Note that (a) and (b) come from rule (1), while the eight transitions of (c) come from rule (2). Also, δ is empty except as defined by (a) through (c). □

Method 2:

Converting CFG to PDA by final State (Suitable for CFG in GNF form)

Algorithm:

Step1: Check whether grammar is in Greibach Normal Form(GNF)

[For GNF, all productions must be of the form, A→aα, where aϵT and αϵV*]

Step2: Push Start symbol of the grammar to the stack and change state from q0 to q1. The transition is:

$\delta(q_0, \epsilon, Z_0)=(q1, SZ_0)$

Step3: For each production of the form, A→aα, where aϵT and αϵV*, add transitions,

$\delta(q_1, a, A)=(q1, \alpha)$

Step4: Finally add a transition to accept the string by final state,

$\delta(q_1, \epsilon, Z_0)=(qf, Z_0)$

Ex: Convert the following grammar to PDA by final state:

S→aSSS | a

Ans:

Step1: The given grammar is in GNF.

Step2: $\delta(q_0, \epsilon, Z_0)=(q1, SZ_0)$

Step3:

S→aSSS	$\delta(q_1, a, S)=(q1, SSS)$
S→a	$\delta(q_0, a, S)=(q1, \epsilon)$

Step4: $\delta(q_1, \epsilon, Z_0)=(qf, Z_0)$

Therefore, $P=(\{ q_0, q_1, q_f \}, \{a\}, \{S, Z_0 \}, \delta, q_0, Z_0, \{q_f\})$

****