Subject: Automata Theory and Computability
Sub Code: 15CS54
Module -III
Context-Free Grammars and Pushdown Automata (PDA)

Course Outcomes-(CO)

At the end of the course student will be able to:

i. Explain core concepts in Automata and Theory of Computation.
ii. Identify different Formal language Classes and their Relationships.
iii. Design Grammars and Recognizers for different formal languages.
iv. Prove or disprove theorems in automata theory using their properties.
v. Determine the decidability and intractability of Computational problems.

Syllabus of Module 3

i. Context-Free Grammars(CFG): Introduction to Rewrite Systems and Grammars
ii. CFGs and languages, designing CFGs,
iii. Simplifying CFGs,
iv. Proving that a Grammar is correct,
v. Derivation and Parse trees, Ambiguity,
vi. Normal Forms.
vii. Pushdown Automata (PDA): Definition of non-deterministic PDA,
viii. Deterministic and Non-deterministic PDAs,
ix. Non-determinism and Halting, Alternative equivalent definitions of a PDA,
x. Alternatives those are not equivalent to PDA.

Text Books:

1. Elaine Rich, Automata, Computability and Complexity, 1st Edition, Pearson Education, 2012/2013. Text Book 1: Ch 11, 12: 11.1 to 11.8, 12.1 to 12.6 excluding 12.3.
2. K L P Mishra, N Chandrasekaran , 3rd Edition, Theory of Computer Science, PHI, 2012

Reference Books:

1. John E Hopcroft, Rajeev Motwani, Jeffery D Ullman, Introduction to Automata Theory, Languages, and Computation, 3rd Edition, Pearson Education, 2013
2. Michael Sipser : Introduction to the Theory of Computation, 3rd edition, Cengage learning,2013
3. John C Martin, Introduction to Languages and The Theory of Computation, 3rd Edition,Tata McGraw –Hill Publishing Company Limited, 2013
4. Peter Linz, “An Introduction to Formal Languages and Automata”, 3rd Edition, Narosa Publishers, 1998
5. Basavaraj S. Anami, Karibasappa K G, Formal Languages and Automata theory, WileyIndia, 2012

Learning Outcomes:
At the end of the module student should be able to:

Sl.No	TLO's
1.	Define context free grammars and languages
2.	Design the grammar for the given context free languages.
3.	Apply the simplification algorithm to simplify the given grammar
4.	Prove the correctness of the grammar
5.	Define leftmost derivation and rightmost derivation
6.	Draw the parse tree to a string for the given grammar.
7.	Define ambiguous and inherently ambiguous grammars.
8.	Prove whether the given grammar is ambiguous grammar or not.
9.	Define Chomsky normal form. Apply the normalization algorithm to convert the grammar to Chomsky normal form.
10.	Define Push down automata (NPDA). Design a NPDA for the given CFG.
11.	Design a DPDA for the given language.
12.	Define alternative equivalent definitions of a PDA.

1. Introduction to Rewrite Systems and Grammars

What is Rewrite System?

A rewrite system (or production system or rule based system) is a list of rules, and an algorithm for applying them. Each rule has a left-hand side and a right hand side.

X Y
(LHS) (RHS)

Examples of rewrite-system rules: $S \rightarrow aSb$, $aS \rightarrow \varepsilon$, $aSb \rightarrow bSabSa$

When a rewrite system R is invoked on some initial string w, it operates as follows:
simple-rewrite(R: rewrite system, w: initial string) =

1. Set working-string to w.
2. Until told by R to halt do:
   - **1.1** Match the LHS of some rule against some part of working-string.
   - **1.2** Replace the matched part of working-string with the RHS of the rule that was matched.
3. Return working-string.

If it returns some string s then R can derive s from w or there exists a derivation in R of s from w.

Examples:

1. A rule is simply a pair of strings where, if the string on the LHS matches, it is replaced by the string on the RHS.
2. The rule $axa \rightarrow aa$ squeeze out whatever comes between a pair of a's.
3. The rule $ab^*ab^*a \rightarrow aaa$ squeeze out b's between a's.

Rewrite systems can be used to define functions. We write rules that operate on an input string to produce the required output string. Rewrite systems can be used to define languages. The rewrite system that is used to define a language is called a grammar.

Grammars Define Languages

A grammar is a set of rules that are stated in terms of two alphabets:

- a terminal alphabet, Σ, that contains the symbols that make up the strings in L(G),
- a nonterminal alphabet, the elements of which will function as working symbols that will be used while the grammar is operating. These symbols will disappear by the time the grammar finishes its job and generates a string.
- A grammar has a unique start symbol, often called S.

A rewrite system formalism specifies:

- The form of the rules that are allowed.
- The algorithm by which they will be applied.
- How its rules will be applied?

Using a Grammar to Derive a String

Simple-rewrite (G, S) will generate the strings in L(G). The symbol ⇒ to indicate steps in a derivation.

Given: S → aS ---- rule 1

S → ε ---- rule 2

A derivation could begin with: S ⇒ aSb ⇒ aaSbb ⇒ aabb

Generating Many Strings

LHS of Multiple rules may match with the working string.

Given: S → aSb ----- rule 1

S → bSa ----- rule 2

S → ε ----- rule 3

Derivation so far: S ⇒ aSb ⇒ aaSbb ⇒

Three are three choices at the next step:

S ⇒ aSb ⇒ aaSbb ⇒ aaaSbbb (using rule 1),

S ⇒ aSb ⇒ aaSbb ⇒ aabSabb (using rule 2),

S ⇒ aSb ⇒ aaSbb ⇒ aabb (using rule 3).

One rule may match in more than one way.

Given: S → aTTb ----- rule 1

T → bTa ----- rule 2

T → ε ----- rule 3

Derivation so far: S ⇒ aTTb ⇒

Two choices at the next step:

S ⇒ aTTb ⇒ abTaTb ⇒

S ⇒ aTTb ⇒ aTbTab ⇒

When to Stop

Case 1: The working string no longer contains any nonterminal symbols (including, when it is ε). In this case, we say that the working string is generated by the grammar.

Example: S ⇒ aSb ⇒ aaSbb ⇒ aabb

Case 2: There are nonterminal symbols in the working string but none of them appears on the left-hand side of any rule in the grammar. In this case, we have a blocked or non-terminated derivation but no generated string.

Given: S → aSb ----- rule 1
S→ bTa ----- rule 2
S → ε ----- rule 3

Derivation so far: S ⇒ aSb ⇒ abTab ⇒

Case 3: It is possible that neither case 1 nor case 2 is achieved.

Given: S → Ba -----rule 1
B → bB -----rule 2

Then all derivations proceed as: S ⇒ Ba ⇒ bBa ⇒ bbBa ⇒ bbbBa ⇒ bbbbBa ⇒ ...

The grammar generates the language Ø.

2. Context –Free Grammar and Languages

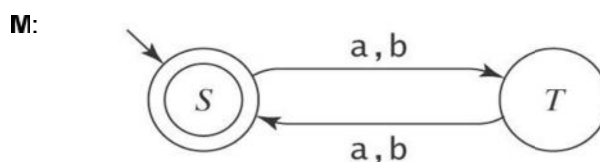
Recall Regular Grammar which has a left-hand side that is a single nonterminal and have a right-hand side that is ε or a single terminal or a single terminal followed by a single nonterminal.

X Y
(NT) (ε or T or T NT)

Example: L = {w Î {a, b}* : |w| is even} RE = ((aa) (ab) (ba) (bb))*

G:
S **→ ε**
S **→ a*T***
S **→ b*T***
T **→ a*S***
T **→ b*S***

M:



Context Free Grammars

X Y
(NT) (No restriction)

No restrictions on the form of the right hand sides. But require single non-terminal on left hand side.

Example: S → ε, S → a, S → T, S → aSb, S→ aSbbT are allowed.
ST→ aSb, a→ aSb, ε → a are not allowed.

The name for these grammars "Context Free" makes sense because using these rules the decision to replace a nonterminal by some other sequence is made without looking at the context in which the nonterminals occurs.

Definition Context-Free Grammar

A context-free grammar G is a quadruple, (V, Σ, R, S), where:

- V is the rule alphabet, which contains nonterminals and terminals.
- Σ (the set of terminals) is a subset of V,
- R (the set of rules) is a finite subset of (V - Σ) ×V*,
- S (the start symbol) is an element of V - Σ.

Given a grammar G, define $x \Rightarrow_G y$ to be the binary relation derives-in-one-step, defined so that $\forall$ x,y ∈ V* ($x \Rightarrow_G y$ iff x = αAβ, y = α γ β and there exists a rule A → γ is in R_G)
Any sequence of the form $w_0 \Rightarrow_G w_1 \Rightarrow_G w_2 \Rightarrow_G \ldots \Rightarrow_G w_n$ is called a derivation in G. Let $\Rightarrow_G{}^*$ be the reflexive, transitive closure of $\Rightarrow_G$. We'll call $\Rightarrow_G{}^*$ the derive relation.

A derivation will halt whenever no rules on the left hand side matches against working-string. At every step, any rule that matches may be chosen.

Language generated by G, denoted L(G), is: L(G) = {w ∈ Σ* : $S \Rightarrow_G{}^* w$}. A language L is context-free iff it is generated by some context-free grammar G. The context-free languages (or CFLs) are a proper superset of the regular languages.

Example: L = A^nB^n = {a^nb^n : n 0} = {ε, ab, aabb, aaabbb, …}
G = {{S, a, b}, {a, b}, R, S}, where: R = { S → aSb , S → ε}
Example derivation in G: S ⇒ aSb ⇒ aaSbb ⇒ aaaSbbb ⇒ aaabbb or S ⇒* aaabbb

Recursive Grammar Rules

A grammar is recursive iff it contains at least one recursive rule. A rule is recursive iff it is $X \rightarrow w_1Yw_2$, where: $Y \Rightarrow^* w_3Xw_4$ for some w_1, w_2, w_3, and w_4 in V*. Expanding a non-terminal according to these rules can eventually lead to a string that includes the same non-terminal again.
Example1: L = A^nB^n = {a^nb^n : n 0} Let G = ({S, a, b}, {a, b}, {S → a S b, S → ε}, S)

Example 2: Regular grammar whose rules are {S → a T, T → a W, W → a S, W → a }

Example 3: The Balanced Parenthesis language
Bal = {w ∈ {),(}*: the parenthesis are balanced} = {ε , (), (()), ()(), (()()) …………….........}
G={{S,),(}, {),(},R,S} where R={ S → ε S → SS S → (S) }
Some example derivations in G:
S ⇒ (S) ⇒ ()
S ⇒ (S) ⇒ (SS) ⇒ ((S)S) ⇒ (() S)) ⇒ (() (S)) ⇒ (()())
So, S ⇒* () and S ⇒* (()())
Recursive rules make it possible for a finite grammar to generate an infinite set of strings.

Self-Embedding Grammar Rules

A grammar is self-embedding iff it contains at least one self-embedding rule. A rule in a grammar G is self-embedding iff it is of the form $X \rightarrow w_1Yw_2$, where $Y \Rightarrow^* w_3Xw_4$ and both w_1w_3 and w_4w_2 are in Σ^+. No regular grammar can impose such a requirement on its strings.

Example:	$S \rightarrow aSa$	is self-embedding
	$S \rightarrow aS$	is recursive but not self- embedding
	$S \rightarrow aT$	
	$T \rightarrow Sa$	is self-embedding

Example : PalEven = $\{ww^R : w \in \{a, b\}^*\}$= The L of even length palindrome of a's and b's.

L = {ε, aa, bb, aaaa, abba, baab, bbbb, ..…….}

G = {{S, a, b}, {a, b}, R, S}, where:

R = { $S \rightarrow aSa$ ----- rule 1
$S \rightarrow bSb$ ----- rule 2
$S \rightarrow \varepsilon$ ----- rule 3 }.

Example derivation in G:

$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abba$

Where Context-Free Grammars Get Their Power

If a grammar G is not self-embedding then L(G) is regular. If a language L has the property that every grammar that defines it is self-embedding, then L is not regular.

More flexible grammar-writing notations

a. Notation for writing practical context-free grammars. The symbol | should be read as "or". It allows two or more rules to be collapsed into one.

Example:

$S \rightarrow a\ S\ b$

$S \rightarrow b\ S\ a$ can be written as *S È a S b | b S a |* ∨

$S \rightarrow \varepsilon$

b. Allow a nonterminal symbol to be any sequence of characters surrounded by angle brackets.

Example1: BNF for a Java Fragment

```
<block> ::= {<stmt-list>} | {}
<stmt-list> ::= <stmt> | <stmt-list> <stmt>
<stmt> ::= <block> | while (<cond>) <stmt> |
           if (<cond>) <stmt> |
           do <stmt> while (<cond>); |
           <assignment-stmt>; |
           return | return <expression> |
           <method-invocation>;
```

Example2: A CFG for C++ compound statements:
<compound stmt> → { <stmt list> }
<stmt list> → <stmt> <stmt list> | epsilon
<stmt> → <compound stmt>
<stmt> → if (<expr>) <stmt>
<stmt> → if (<expr>) <stmt> else <stmt>
<stmt> → while (<expr>) <stmt>
<stmt> → do <stmt> while (<expr>) ;
<stmt> → for (<stmt> <expr> ; <expr>) <stmt>
<stmt> → case <expr> : <stmt>
<stmt> → switch (<expr>) <stmt>
<stmt> → break ; | continue ;
<stmt> → return <expr> ; | goto <id> ;

Example3: A Fragment of English Grammar
Notational conventions used are

- Nonterminal = whose first symbol is an uppercase letter
- NP = derive noun phrase
- VP = derive verb phrase

S → NP VP
NP → the Nominal | a Nominal | Nominal | ProperNoun | NP PP
Nominal → N | Adjs N
N → cat | dogs | bear | girl | chocolate | rifle
ProperNoun → Chris | Fluffy
Adjs → Adj Adjs | Adj
Adj → young | older | smart
VP → V | V NP | VP PP
V → like | likes | thinks | shots | smells
PP → Prep NP
Prep → with

3. Designing Context-Free Grammars

If L has a property that every string in it has two regions & those regions must bear some relationship to each other, then the two regions must be generated in tandem. Otherwise, there is no way to enforce the necessary constraint.

Example 1: $L = \{a^n b^n c^m : n, m \quad 0\} = L = \{\varepsilon, ab, c, abc, abcc, aabbc, \ldots\ldots.\}$

The c^m portion of any string in L is completely independent of the $a^n b^n$ portion, so we should generate the two portions separately and concatenate them together.

G = ({S, A, C, a, b, c}, {a, b, c}, R, S} where:

R = { $S \rightarrow AC$ /* generate the two independent portions

$A \rightarrow aAb \mid \varepsilon$ /* generate the $a^n b^n$ portion, from the outside in

$C \rightarrow cC \mid \varepsilon$ } /* generate the c^m portion

Example derivation in G for string abcc:

$S \Rightarrow AC \Rightarrow aAbC \Rightarrow abC \Rightarrow abcC \Rightarrow abccC \Rightarrow abcc$

Example 2: $L=\{ a^i b^j c^k : j=i+k, i, k \quad 0\}$ on substituting $j=i+k \Rightarrow L = \{a^i b^i b^k c^k : i, k \quad 0\}$

$L = \{\varepsilon, abbc, aabbbbcc, abbbcc \ldots\ldots.\}$

The $a^i b^i$ portion of any string in L is completely independent of the $b^k c^k$ portion, so we should generate the two portions separately and concatenate them together.

G = ({S, A, B, a, b, c}, {a, b, c}, R, S} where:

R = { $S \rightarrow AB$ /* generate the two independent portions

$A \rightarrow aAb \mid \varepsilon$ /* generate the $a^i b^i$ portion, from the outside in

$B \rightarrow bBc \mid \varepsilon$ } /* generate the $b^k c^k$ portion

Example derivation in G for string abbc:

$S \Rightarrow AB \Rightarrow aAbB \Rightarrow abB \Rightarrow abbBc \Rightarrow abbc$

Example 3: $L=\{ a^i b^j c^k : i=j+k, j, k \quad 0\}$ on substituting $i=j+k \Rightarrow L = \{a^k a^j b^j c^k : j, k \quad 0\}$

$L = \{\varepsilon, ac, ab, aabc, aaabcc, \ldots\ldots.\}$

The $a^i b^i$ is the inner portion and $a^k c^k$ is the outer portion of any string in L.

G = ({S, A, a, b, c}, {a, b, c}, R, S} where:

R = { $S \rightarrow aSc \mid A$ /* generate the $a^k c^k$ outer portion

$A \rightarrow aAb \mid \varepsilon$ /* generate the $a^j b^j$ inner portion}

Example derivation in G for string aabc:

$S \Rightarrow aSc \Rightarrow aAc \Rightarrow aaAbc \Rightarrow aabc$

Example 4: $L = \{a^n w w^R b^n : w \in \{a, b\}^*\} = \{\varepsilon, ab, aaab, abbb, aabbab, aabbbbab, \ldots\ldots\ldots\}$

The $a^n b^n$ is the inner portion and ww^R is the outer portion of any string in L.

G = {{S, A, a, b}, {a, b}, R, S}, where:

R = { $S \rightarrow aSb$ ----- rule 1

$S \rightarrow A$ ----- rule 2

$A \rightarrow aAa$ ----- rule 3

$A \rightarrow bAb$ ----- rule 4

$A \rightarrow \varepsilon$ ----- rule 5 }.

Example derivation in G for string aabbab:

$S \Rightarrow aSb \Rightarrow aAb \Rightarrow aaAab \Rightarrow aabAbab \Rightarrow aabbab$

Example 5: Equal Numbers of a's and b's. = $\{w \in \{a, b\}^*: \#_a(w) = \#_b(w)\}$.

L = {ε, ab, ba, abba, aabb, baba, bbaa, …….}

G = {{S, a, b}, {a, b}, R, S}, where:

R = { $S \rightarrow aSb$ ----- rule 1

$S \rightarrow bSa$ ----- rule 2

$S \rightarrow SS$ ----- rule 3

$S \rightarrow \varepsilon$ ----- rule 4 }.

Example derivation in G for string abba:

$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abba$

Example 6

L = $\{a^i b^j : 2i = 3j + 1\} = \{a^2 b^1, a^5 b^3, a^8 b^5 \ldots\ldots\}$

G = {{S, a, b}, {a, b}, R, S}, where:

$a^i b^j$	$2i = 3j + 1$
$a^2 b^1$	$2*2= 3*1 + 1 = 4$
$a^5 b^3$	$2*5= 3*3 + 1 = 10$
$a^8 b^5$	$2*8= 3*5 + 1 = 16$

R={ $S \rightarrow aaaSbb \mid aab$ }

Example derivation in G for string aaaaabbb:

$S \Rightarrow aaaSbb \Rightarrow aaaaabbb$

4. Simplifying Context-Free Grammars

Two algorithms used to simplify CFG

a. To find and remove unproductive variables using removeunproductive(G:CFG)
b. To find and remove unreachable variables using removeunreachable(G:CFG)

a. Removing Unproductive Nonterminals:

Removeunproductive (G: CFG) =

1. G′ = G.
2. Mark every nonterminal symbol in G′ as unproductive.
3. Mark every terminal symbol in G′ as productive.
4. Until one entire pass has been made without any new symbol being marked do:

   For each rule $X \rightarrow \alpha$ in R do:

   If every symbol in α has been marked as productive and X has not yet been marked as productive then:

   Mark X as productive.
5. Remove from G′ every unproductive symbol.
6. Remove from G′ every rule that contains an unproductive symbol.
7. Return G′.

Example: G = ({S, A, B, C, D, a, b}, {a, b}, R, S), where

R = { $S \rightarrow AB \mid AC$

$A \rightarrow aAb \mid \varepsilon$

$B \rightarrow bA$

$C \rightarrow bCa$

$D \rightarrow AB$ }

1) a and b terminal symbols are productive
2) A is productive(because $A \rightarrow aAb$)
3) B is productive(because $B \rightarrow bA$)
4) S & D are productive(because $S \rightarrow AB$ & $D \rightarrow AB$)
5) C is unproductive

On eliminating C from both LHS and RHS the rule set R′ obtained is

R′ = { $S \rightarrow AB$ $A \rightarrow aAb \mid \varepsilon$ $B \rightarrow bA$ $D \rightarrow AB$ }

b. Removing Unreachable Nonterminals

Removeunreachable (G: CFG) =

1. G′ = G.
2. Mark S as reachable.
3. Mark every other nonterminal symbol as unreachable.
4. Until one entire pass has been made without any new symbol being marked do:

   For each rule $X \rightarrow \alpha A \beta$ (where $A \in V - \Sigma$) in R do:

   If X has been marked as reachable and A has not then:

   Mark A as reachable.
5. Remove from G′ every unreachable symbol.
6. Remove from G′ every rule with an unreachable symbol on the left-hand side.
7. Return G′.

Example

G = ({S, A, B, C, D, a, b}, {a, b}, R, S), where

R′ = { $S \rightarrow AB$

$A \rightarrow aAb \mid \varepsilon$

$B \rightarrow bA$

$D \rightarrow AB$ }

S, A, B are reachable but D is not reachable, on eliminating D from both LHS and RHS the rule set R″ is

R″ = { $S \rightarrow AB$

$A \rightarrow aAb \mid \varepsilon$

$B \rightarrow bA$ }

5. Proving the Correctness of a Grammar

Given some language L and a grammar G, can we prove that G is correct (ie it generates exactly the strings in L)

To do so, we need to prove two things:

1. Prove that G generates only strings in L.
2. Prove that G generates all the strings in L.

6. Derivations and Parse Trees

Algorithms used for generation and recognition must be systematic. The expansion order is important for algorithms that work with CFG. To make it easier to describe such algorithms, we define two useful families of derivations.

a. A leftmost derivation is one in which, at each step, the leftmost nonterminal in the working string is chosen for expansion.
b. A rightmost derivation is one in which, at each step, the rightmost nonterminal in the working string is chosen for expansion.

Example 1 : S   AB | aaB   A   a | Aa   B   b
Left-most derivation for string aab is   S ⇒ AB ⇒ AaB ⇒ aaB ⇒ aab
Right-most derivation for string aab is   S ⇒ AB ⇒ Ab ⇒ Aab ⇒ aab

Example 2: S→iCtS | iCtSeS | x   C→y
Left-most Derivation for string iytiytxex is S ⇒ iCtS ⇒ iytS ⇒ iytiCtSeS ⇒ iytiytSeS ⇒ iytiytxe ⇒ iytiytxex
Right-most Derivation for string iytiytxex is S ⇒ iCtSeS ⇒ iCtSex ⇒ iCtiCtSex ⇒ iCtiCtxex ⇒ iCtiytxex ⇒ iytiytxex

Example 3: A Fragment of English Grammar are

S → NP VP

NP → the Nominal | a Nominal | Nominal | ProperNoun | NP PP

Nominal → N | Adjs N

N → cat | dogs | bear | girl | chocolate | rifle

ProperNoun → Chris | Fluffy

Adjs → Adj Adjs | Adj

Adj → young | older | smart

VP → V | V NP | VP PP

V → like | likes | thinks | shots | smells

PP → Prep NP

Prep → with

Left-most Derivation for the string “the smart cat smells chocolate”

S ⇒ NP VP
⇒ the Nominal VP
⇒ the Adjs N VP
⇒ the Adj N VP
⇒ the smart N VP
⇒ the smart cat VP
⇒ the smart cat V NP
⇒ the smart cat smells NP
⇒ the smart cat smells Nominal
⇒ the smart cat smells N
⇒ the smart cat smells chocolate

Right-most Derivation for the string “the smart cat smells chocolate”

S ⇒ NP VP
⇒ NP V NP
⇒ NP V Nominal
⇒ NP V N
⇒ NP V chocolate
⇒ NP smells chocolate
⇒ the Nominal smells chocolate
⇒ the Adjs N smells chocolate
⇒ the Adjs cat smells chocolate
⇒ the Adj cat smells chocolate
⇒ the smart cat smells chocolate

Parse Trees

Regular grammar: in most applications, we just want to describe the set of strings in a language. Context-free grammar: we also want to assign meanings to the strings in a language, for which we care about internal structure of the strings. Parse trees capture the essential grammatical structure of a string. A program that produces such trees is called a parser. A parse tree is an (ordered, rooted) tree that represents the syntactic structure of a string according to some formal grammar. In a parse tree, the interior nodes are labeled by non terminals of the grammar, while the leaf nodes are labeled by terminals of the grammar or ε.

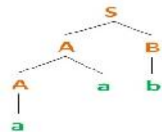
A parse tree, derived by a grammar G = (V, S, R, S), is a rooted, ordered tree in which:

1. Every leaf node is labeled with an element of ∪{ ε },
2. The root node is labeled S,
3. Every other node is labeled with some element of: V – , and
4. If m is a nonleaf node labeled X and the children of m are labeled $x_1, x_2, \ldots, x_n$, then R contains the rule X $x_1, x_2, \ldots, x_n$.

Example 1: S → AB | aaB   A → a | Aa   B → b

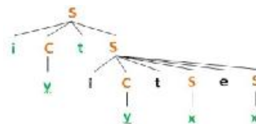
Left-most derivation for the string aab is S ⇒ AB ⇒ AaB ⇒ aaB ⇒ aab

Parse tree obtained is

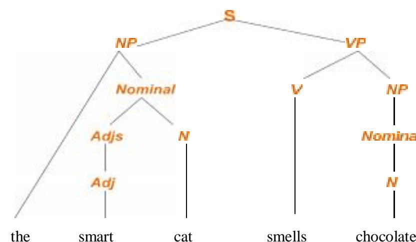


Example 2: S→iCtS | iCtSeS | x   C→y

Left-most Derivation for string iytiytxex isS ⇒ iCtS ⇒ iytS ⇒ iytiCtSeS ⇒ iytiytSeS ⇒ iytiytxeS ⇒ iytiytxex



Example 3: Parse Tree -Structure in English for the string "the smart cat smells chocolate". It is clear from the tree that the sentence is not about cat smells or smart cat smells.



A parse tree may correspond to multiple derivations. From the parse tree, we cannot tell which of the following is used in derivation:

S ⇒ NP VP ⇒ the Nominal VP ⇒

S ⇒ NP VP ⇒ NP V NP ⇒

Parse trees capture the important structural facts about a derivation but throw away the details of the nonterminal expansion order. The order has no bearing on the structure we wish to assign to a string.

Generative Capacity

Because parse trees matter, it makes sense, given a grammar G, to distinguish between:

1. G's weak generative capacity, defined to be the set of strings, L(G), that G generates, and
2. G's strong generative capacity, defined to be the set of parse trees that G generates.

When we design grammar, it will be important that we consider both their weak and their strong generative capacities.

7. Ambiguity

Sometimes a grammar may produce more than one parse tree for some (or all) of the strings it generates. When this happens we say that the grammar is ambiguous. A grammar is ambiguous iff there is at least one string in L(G) for which G produces more than one parse tree.

Example 1: Bal={w ∈ {),(}*: the parenthesis are balanced}.

G={{S,),(}, {),(},R,S} where R={ S → ε S → SS S → (S) }

Left-most Derivation1 for the string (())() is S ⇒ S⇒(S)S ⇒ ((S))S ⇒ (())S ⇒ (())(S) ⇒ (())()

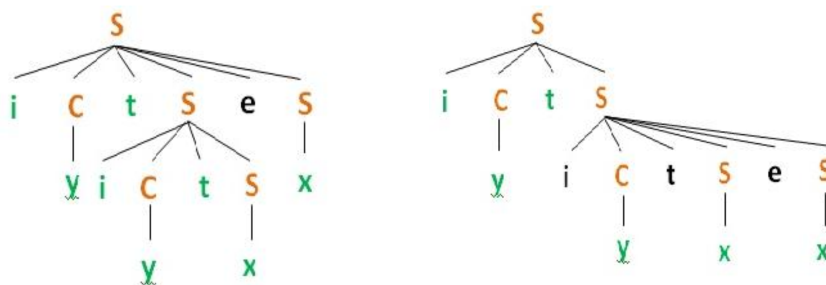
Left-most Derivation2 for the string (())() is S ⇒ SS ⇒SSS ⇒SS ⇒ (S)S ⇒ ((S))S ⇒ (())S ⇒ (())(S) ⇒ (())()

Since both the parse trees obtained for the same string (())() are different, the grammar is ambiguous.

Example 2: S→ iCtS | iCtSeS | x   C→ y

Left-most Derivation for the string iytiytxex is S ⇒ iCtS ⇒ iytS ⇒ iytiCtSeS ⇒ iytiytSeS ⇒ iytiytxeS ⇒ iytiytxex

Right-most Derivation for the string iytiytxex is S ⇒ iCtSeS ⇒ iCtSex ⇒ iCtiCtSex ⇒iCtiCtxex ⇒ iCtiytxex ⇒ iytiytxex

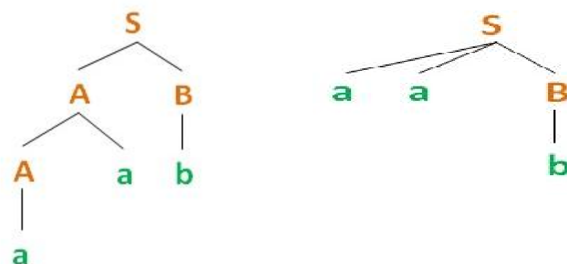


Since both the parse trees obtained for the same string iytiytxex are different, the grammar is ambiguous.

Example 3: S → AB | aaB   A → a | Aa   B → b

Left-most derivation for string aab is   S ⇒ AB ⇒ AaB ⇒ aaB ⇒ aab

Right-most derivation for string aab is   S ⇒ aaB ⇒ aab



Since both the parse trees obtained for the same string aab are different, the grammar is ambiguous.

Why Is Ambiguity a Problem?

With regular languages, for most applications, we do not care about assigning internal structure to strings.

With context-free languages, we usually do care about internal structure because, given a string w, we want to assign meaning to w. It is generally difficult, if not impossible, to assign a unique meaning without a unique parse tree. So an ambiguous G, which fails to produce a unique parse tree is a problem.

Example : Arithmetic Expressions

G = (V, Σ, R, E), where

V = {+, *, (,), id, E},

Σ = {+, *, (,), id},

R = {E → E + E, E → E ∗ E, E → (E), E → id }

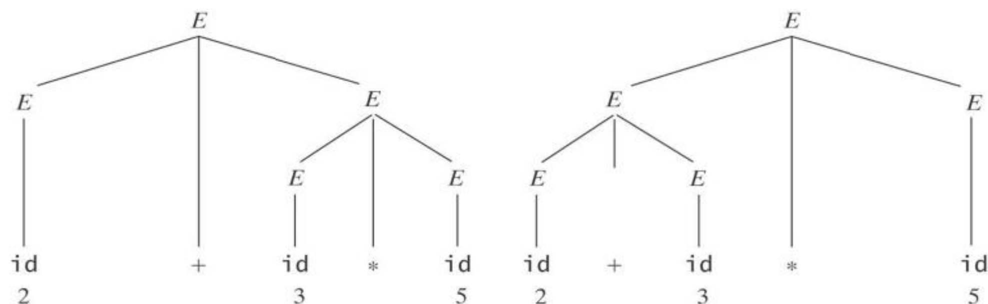
Consider string 2+3*5 written as id +id*id, left-most derivation for string id +id*id is

E ⇒ E*E ⇒ E+E*E ⇒ id+E*E ⇒ id+id*E ⇒ id+id*id.

Similarly the right-most derivation for string id +id*id is

E ⇒ E+E ⇒ E+E*E ⇒ E+E*id ⇒ E+id*id ⇒ id+id*id.

The parse trees obtained for both the derivations are:-



Should the evaluation of this expression return 17 or 25? Designers of practical languages must be careful that they create languages for which they can write unambiguous grammars.

Techniques for Reducing Ambiguity

No general purpose algorithm exists to test for ambiguity in a grammar or to remove it when it is found. But we can reduce ambiguity by eliminating

a. ε rules like S ε
b. Rules with symmetric right-hand sides
   - A grammar is ambiguous if it is both left and right recursive.
   - Fix: remove right recursion
   - S SS or E E + E
c. Rule sets that lead to ambiguous attachment of optional postfixes.

a. Eliminating ∨-Rules

Let G =(V, Σ, R, S) be a CFG. The following algorithm constructs a G′ such that L(G′) = L(G)-{ε} and G′ contains no ε rules:

removeEps(G: CFG) =

1. Let G′ = G.
2. Find the set N of nullable variables in G′.
3. Repeat until G′ contains no modifiable rules that haven't been processed:
   Given the rule $P \rightarrow \alpha Q \beta$, where $Q \in N$, add the rule $P \rightarrow \alpha\beta$ if it is not already present and if $\alpha\beta \neq \varepsilon$ and if $P \neq \alpha\beta$.
4. Delete from G′ all rules of the form $X \rightarrow \varepsilon$.
5. Return G′.

Nullable Variables & Modifiable Rules

A variable X is nullable iff either:

(1) there is a rule $X \rightarrow \varepsilon$, or

(2) there is a rule X → PQR… and P, Q, R, … are all nullable.

So compute N, the set of nullable variables, as follows:

2.1. Set N to the set of variables that satisfy (1).

2.2. Until an entire pass is made without adding anything to N do

Evaluate all other variables with respect to (2).

If any variable satisfies (2) and is not in N, insert it.

A rule is modifiable iff it is of the form: $P \rightarrow \alpha Q \beta$, for some nullable Q.

Example: G = {{S, T, A, B, C, a, b, c}, {a, b, c}, R, S),

R = {S → aTa   T → ABC   A → aA | C   B → Bb | C   C → c | ε }

Applying removeEps

Step2: N = { C }

Step2.2 pass1: N = { A, B, C }

Step2.2 pass2: N = { A, B, C, T }

Step2.2 pass3: no new element found.

Step2: halts.

Step3: adds the following new rules to G′.

{ S → aa

T → AB | BC | AC | A | B | C

A → a

B → b }

The rules obtained after eliminating ε-rules :

{ S → aTa | aa

T → ABC | AB | BC | AC | A | B | C

A → aA | C | a

B → Bb | C | b

C → c }

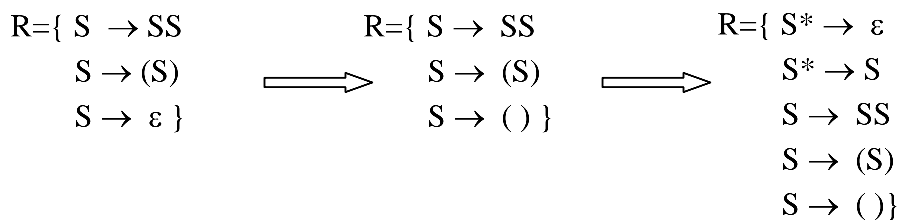
What If ∨ è L?

Sometimes L(G) contains ε and it is important to retain it. To handle this case the algorithm used is

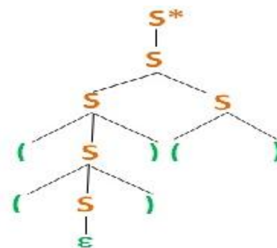
atmostoneEps(G: CFG) =

1. G″ = removeEps(G).
2. If S_G is nullable then /* i. e., ε ∈ L(G)
   2.1 Create in G″ a new start symbol S*.
   2.2 Add to $R_{G''}$ the two rules: $S^* \rightarrow \varepsilon$ and $S^* \rightarrow S_G$.
3. Return G″.

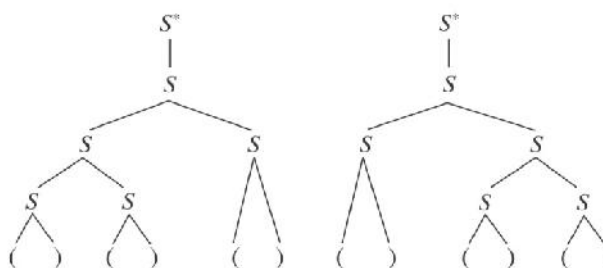
Example: Bal={w ∈ {),(}*: the parenthesis are balanced}.



The new grammar built is better than the original one. The string (())() has only one parse tree.



But it is still ambiguous as the string ()()() has two parse trees?



Replace $S \rightarrow SS$ with one of:

$S \rightarrow S\, S_1$ /* force branching to the left

$S \rightarrow S_1 S$ /* force branching to the right

So we get:

$S^* \rightarrow \varepsilon \mid S$

$S \rightarrow SS_1$ /* force branching only to the left

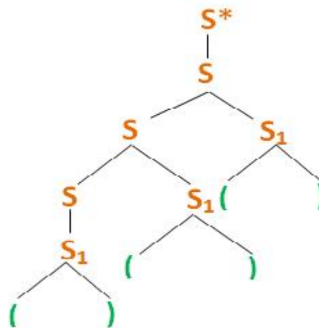
$S \rightarrow S_1$ /* add rule

$S_1 \rightarrow (S) \mid ()$

Unambiguous Grammar for Bal={w ∈ {),(}*: the parenthesis are balanced}.
G={{S,),(}, {),(},R,S} where

$$S^* \rightarrow \varepsilon_n \mid S$$
$$S \rightarrow SS_1 \mid S_1$$
$$S_1 \rightarrow (S) \mid ()$$

The parse tree obtained for the string ()()() is



Unambiguous Arithmetic Expressions
Grammar is ambiguous in two ways:

a. It fails to specify associatively.

Ex: there are two parses for the string id + id + id, corresponding to the bracketing (id + id) + id and id + (id + id)

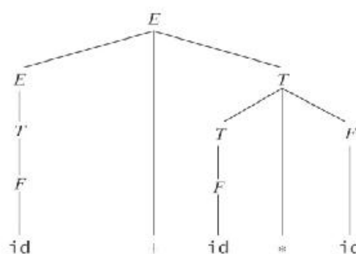
b. It fails to define a precedence hierarchy for the operations + and *.

Ex: there are two parses for the string id + id * id, corresponding to the bracketing (id + id)* id and id + (id * id)

The unambiguous grammar for the arithmetic expression is:

$$E \rightarrow E + T$$
$$E \rightarrow T$$
$$T \rightarrow T * F$$
$$T \rightarrow \ F$$
$$F \rightarrow (E)$$
$$F \rightarrow id$$

For identical operators: forced branching to go in a single direction (to the left). For precedence Hierarchy: added the levels T (for term) and F (for factor)
The single parse tree obtained from the unambiguous grammar for the arithmetic expression is:



Proving that the grammar is Unambiguous

A grammar is unambiguous iff for all strings w, at every point in a leftmost derivation or rightmost derivation of w, only one rule in G can be applied.

$S^* \rightarrow \varepsilon$ ---(1)

$S^* \rightarrow S$ ---(2)

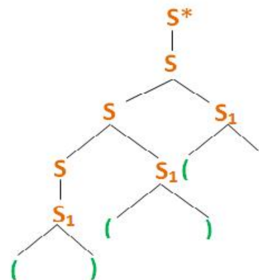
$S \rightarrow SS_1$ ---(3)

$S \rightarrow S_1$ ---(4)

$S_1 \rightarrow (S)$ ---(5)

$S_1 \rightarrow ()$ ---(6)

$S^* \Rightarrow S \Rightarrow SS_1 \Rightarrow SS_1S_1 \Rightarrow S_1S_1S_1 \Rightarrow ()\, S_1S_1 \Rightarrow ()\,()\, S_1 \Rightarrow ()\,()\,()$



Inherent Ambiguity

In many cases, for an ambiguous grammar G, it is possible to construct a new grammar G′ that generate L(G) with less or no ambiguity. However, not always. Some languages have the property that every grammar for them is ambiguous. We call such languages inherently ambiguous.

Example: $L = \{a^ib^jc^k : i, j, k \geq 0, i=j \text{ or } j=k\}$.

Every string in L has either (or both) the same number of a's and b's or the same number of b's and c's. $L = \{a^nb^nc^m : n, m \geq 0\} \cup \{a^nb^mc^m : n, m \geq 0\}$.

One grammar for L has the rules:

$S \rightarrow S_1 \mid S_2$

$S_1 \rightarrow S_1c \mid A$ /* Generate all strings in $\{a^nb^nc^m\}$.

$A \rightarrow aAb \mid \varepsilon$

$S_2 \rightarrow aS_2 \mid B$ /* Generate all strings in $\{a^nb^mc^m\}$.

$B \rightarrow bBc \mid \varepsilon$

Consider the string $a^2b^2c^2$.

It has two distinct derivations, one through S_1 and the other through S_2

$S \Rightarrow S_1 \Rightarrow S_1c \Rightarrow S_1cc \Rightarrow Acc \Rightarrow aAbcc \Rightarrow aaAbbcc \Rightarrow aabbcc$

$S \Rightarrow S_2 \Rightarrow aS_2 \Rightarrow aaS_2 \Rightarrow aaB \Rightarrow aabBc \Rightarrow aabbBcc \Rightarrow aabbcc$

Given any grammar G that generates L, there is at least one string with two derivations in G.

Both of the following problems are undecidable:

- Given a context-free grammar G, is G ambiguous?
- Given a context-free language L, is L inherently ambiguous

8. Normal Forms

We have seen in some grammar where RHS is ε , it makes grammar harder to use. Lets see what happens if we carry the idea of getting rid of ε -productions a few steps farther. To make our tasks easier we define normal forms.

Normal Forms - When the grammar rules in G satisfy certain restrictions, then G is said to be in Normal Form.

- Normal Forms for queries & data can simplify database processing.
- Normal Forms for logical formulas can simplify automated reasoning in AI systems and in program verification system.
- It might be easier to build a parser if we could make some assumptions about the form of the grammar rules that the parser will use.

Normal Forms for Grammar

Among several normal forms, two of them are:-

- Chomsky Normal Form(CNF)
- Greibach Normal Form(GNF)

Chomsky Normal Form (CNF)

In CNF we have restrictions on the length of RHS and the nature of symbols on the RHS of the grammar rules.

A context-free grammar G = (V, , R, S) is said to be in Chomsky Normal Form (CNF), iff every rule in R is of one of the following forms:

$X \rightarrow a$ where $a \in \Sigma$, or

$X \rightarrow BC$ where B and C $\in$ V-Σ

Example: $S \rightarrow AB$, $A \rightarrow a, B \rightarrow b$

Every parse tree that is generated by a grammar in CNF has a branching factor of exactly 2 except at the branches that leads to the terminal nodes, where the branching factor is 1.

Using this property parser can exploit efficient data structure for storing and manipulating binary trees. Define straight forward decision procedure to determine whether w can be generated by a CNF grammar G. Easier to define other algorithms that manipulates grammars.

Greibach Normal Form (GNF)

GNF is a context free grammar G = (V, Σ, R, S), where all rules have one of the following forms: $X \rightarrow a\beta$ where $a \in \Sigma$ and $\beta \in (V\text{-}\Sigma)^*$

Example: $S \rightarrow aA \mid aAB$, $A \rightarrow a, B \rightarrow b$

In every derivation precisely one terminal is generated for each rule application. This property is useful to define a straight forward decision procedure to determine whether w can be generated by GNF grammar G. GNF grammars can be easily converted to PDA with no ε transitions.

Converting to Chomsky Normal Form

Apply some transformation to G such that the language generated by G is unchanged.

1. **1.** Rule Substitution.

   Example: $X \to aYc$ $Y \to b$ $Y \to ZZ$ equivalent grammar constructed is $X \to abc \mid aZZc$

There exists 4-steps algorithm to convert a CFG G into a new grammar G_c such that: $L(G) = L(G_c) - \{\varepsilon\}$

convertChomsky(G:CFG) =

1. G' = removeEps(G:CFG) $S \to \varepsilon$
2. G'' = removeUnits(G':CFG) $A \to B$
3. G''' = removeMixed(G'':CFG) $A \to aB$
4. G^{IV} = removeLong(G''' :CFG) $S \to ABCD$

return G_c

Remove Epsilon using removeEps(G:CFG)

Find the set N of nullable variables in G.

X is nullable iff either $X \to \varepsilon$ or $(X \to A,\ A \to \varepsilon) : X \to \varepsilon$

Example1: G: $S \to aACa$

$A \to B \mid a$

$B \to C \mid c$

$C \to cC \mid \varepsilon$

Now, since $C \to \varepsilon$, C is nullable

since $B \to C$, B is nullable

since $A \to B$, A is nullable

Therefore N = { A,B,C}

removeEps returns G':

$S \to aACa \mid aAa \mid aCa \mid aa$

$A \to B \mid a$

$B \to C \mid c$

$C \to cC \mid c$

Remove Unit Productions using removeUnits(G:CFG)

Unit production is a rule whose right hand side consists of a single nonterminal symbol.

Ex: $A \to B$. Remove all unit production from G'.

Consider the remaining rules of G'.

G′: $S \to aACa \mid aAa \mid aCa \mid aa$

$A \to B \mid a$

$B \to C \mid c$

$C \to cC \mid c$

Remove $A \to B$ But $B \to C \mid c$, so Add $A \to C \mid c$

Remove $B \to C$ Add $B \to cC$ ($B \to c$, already there)

Remove $A \to C$ Add $A \to cC$ ($A \to c$, already there)

removeUnits returns G'' :

$S \rightarrow aACa \mid aAa \mid aCa \mid aa$

$A \rightarrow cC \mid a \mid c$

$B \rightarrow cC \mid c$

$C \rightarrow cC \mid c$

Remove Mixed using removeMixed(G'':CFG)

Mixed is a rule whose right hand side consists of combination of terminals or terminals with nonterminal symbol. Create a new nonterminal T_a for each terminal $a \in \Sigma$. For each T_a, add the rule $T_a \rightarrow a$

Consider the remaining rules of G'' :

$S \rightarrow aACa \mid aAa \mid aCa \mid aa$

$A \rightarrow cC \mid a \mid c$

$B \rightarrow cC \mid c$

$C \rightarrow cC \mid c$

removeMixed returns G''' :

$S \rightarrow T_aACT_a \mid T_aAT_a \mid T_aCT_a \mid T_aT_a$

$A \rightarrow T_cC \mid a \mid c$

$B \rightarrow T_cC \mid c$

$C \rightarrow T_cC \mid c$

$T_a \rightarrow a$

$T_c \rightarrow c$

Remove Long using removeLong(G''' :CFG)

Long is a rule whose right hand side consists of more than two nonterminal symbol.

R: $A \rightarrow BCDE$ is replaced as: $A \rightarrow BM_2$

$M_2 \rightarrow CM_3$

$M_3 \rightarrow DE$

Consider the remaining rules of G''':

$S \rightarrow T_aACT_a \mid T_aAT_a \mid T_aCT_a$

Now, by applying removeLong we get :

$S \rightarrow T_aS_1$

$S_1 \rightarrow AS_2$

$S_2 \rightarrow CT_a$

$S \rightarrow T_aS_3$

$S_3 \rightarrow AT_a$

$S \rightarrow T_aS_2$

Now, by apply removeLong returns G'^{v} :

$S \to T_aS_1 \mid T_aS_3 \mid T_aS_2 \mid T_aT_a$

$S_1 \to AS_2$

$S_2 \to CT_a$

$S_3 \to AT_a$

$A \to T_cC \mid a \mid c$

$B \to T_cC \mid c$

$C \to T_cC \mid c$

$T_a \to a$

$T_c \to c$

Example 2: Apply the normalization algorithm to convert the grammar to CNF

G: $S \to aSa \mid B$

$B \to bbC \mid bb$

$C \to cC \mid \varepsilon$

removeEps(G:CFG) returns

G′: $S \to aSa \mid B$

$B \to bbC \mid bb$

$C \to cC \mid c$

removeUnits(G':CFG) returns

G″ : $S \to aSa \mid bbC \mid bb$

$B \to bbC \mid bb$

$C \to cC \mid c$

removeMixed(G'':CFG) returns

G''': $S \to T_aST_a \mid T_bT_bC \mid T_bT_b$

$B \to T_bT_bC \mid T_bT_b$

$C \to T_cC \mid c$

$T_a \to a$

$T_b \to b$

$T_c \to c$

removeLong(G''' :CFG) returns

G'^{v}: $S \to T_a S_1 \mid T_b S_2 \mid T_bT_b$

$S_1 \to S\ T_a$

$S_2 \to T_b\ C$

$B \to T_b S_2 \mid T_bT_b$

$C \to T_cC \mid c$

$T_a \to a$

$T_b \to b$

$T_c \to c$

Example 3: Apply the normalization algorithm to convert the grammar to CNF

G: S ABC
A aC | D
B bB | | A
C Ac | | Cc
D aa

removeEps(G:CFG) returns

G′: S ABC | AC | AB | A
A aC | D | a
B bB | A | b
C Ac | Cc | c
D aa

removeUnits(G':CFG) returns

G′′ : S ABC | AC | AB | aC | aa | a
A aC | aa | a
B bB | aC | aa | a | b
C Ac | Cc | c
D aa

removeMixed(G'':CFG) returns

G′′′ : S ABC | AC | AB | T_a C | T_a T_a | a
A T_a C | T_a T_a | a
B T_b B | T_a C | T_a T_a | a | b
C A T_c | C T_c | c
D T_a T_a
$T_a \rightarrow$ a
$T_b \rightarrow$ b
$T_c \rightarrow$ c

removeLong(G''' :CFG) returns

G′^{v}: S AS_1 | AC | AB | T_a C | T_a T_a | a
S_1 BC
A T_a C | T_a T_a | a
B T_b B | T_a C | T_a T_a | a | b
C A T_c | C T_c | c
D T_a T_a
$T_a \rightarrow$ a
$T_b \rightarrow$ b
$T_c \rightarrow$ c

9. Pushdown Automata

An acceptor for every context-free language. A pushdown automata , or PDA, is a finite state machine that has been augmented by a single stack.

Definition of a (NPDA) Pushdown Automaton

M = (K, S, G, , s, A), where:

- K is a finite set of states,
- S is the input alphabet,
- G is the stack alphabet,
- s ∈ K is the initial state,
- A ⊆ K is the set of accepting states, and
- is the transition relation.

Δ is the transition relation. It is a finite subset of

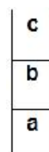
$$(K \times (\Sigma \cup \{\varepsilon\}) \times \Gamma^*) \times (K \times \Gamma^*)$$

state | input or ε | string of symbols to pop from top of stack | state | string of symbols to push on top of stack

Configuration

A configuration of PDA M is an element of K X S* X G*. Current state, Input that is still left to read and, Contents of its stack.

The initial configuration of a PDA M, on input w, is (s, w, ε).



will be written as cba

If $s_1s_2 \ldots s_n$ is pushed onto the stack: the value after the push is $s_1s_2 \ldots s_n$cba

Yields-in-one-step

Yields-in-one-step written $|\!-_M$ relates configuration$_1$ to configuration$_2$ iff M can move from configuration$_1$ to configuration$_2$ in one step. Let c be any element of U {ε }, let $\gamma 1$, $\gamma 2$ and γ be any elements of G*, and let w be any element of S*. Then:

$$(q_1, cw, \gamma 1\gamma) \;|\!-_M (q_2, w, \gamma 2\gamma) \text{ iff } ((q_1, c, \gamma 1), (q_2, \gamma 2)) \in \ .$$

The relation yields, written $|\!-_M$* is the reflexive, transitive closure of $|\!-_M$ C_1 yields configuration C_2 iff C_1 $|\!-_M$* C_2

Computation

A computation by M is a finite sequence of configurations $C_0, C_1, C_2, \ldots, C_n$ for some $n \geq 0$ such that:

- C_0 is an initial configuration,
- C_n is of the form (q, ε, γ), for some $q \in K$ and some string γ in G*, and
- $C_0 \vdash_M C_1 \vdash_M C_2 \vdash_M \ldots \vdash_M C_n$.

Nondeterminism

If M is in some configuration (q_1, s, γ) it is possible that:

contains exactly one transition that matches. In that case, M makes the specified move.

contains more than one transition that matches. In that case, M chooses one of them.

contains no transition that matches. In that case, the computation that led to that configuration halts.

Accepting

Let C be a computation of M on input w then C is an accepting configuration

iif $C = (s, w, \varepsilon) \vdash_M^* (q, \varepsilon, \varepsilon)$, for some $q \in A$.

A computation accepts only if it runs out of input when it is in an accepting state and the stack is empty.

C is a rejecting configuration iif $C = (s, w, \varepsilon) \vdash_M^* (q, w', \alpha)$,

where C is not an accepting computation and where M has no moves that it can makes from (q, w', α). A computation can reject only if the criteria for accepting have not been met and there are no further moves that can be taken.

Let w be a string that is an element of S* . Then:

- M accepts w iif atleast one of its computations accepts.
- M rejects w iif all of its computations reject.

The language accepted by M, denoted L(M), is the set of all strings accepted by M. M rejects a string w iff all paths reject it.

It is possible that, on input w, M neither accepts nor rejects. In that case, no path accepts and some path does not reject.

Transition

Transition $((q_1, c, \gamma 1), (q_2, \gamma 2))$ says that "If c matches the input and g_1 matches the current top of the stack, the transition from q_1 to q_2 can be taken. Then, c will be removed from the input, $\gamma 1$ will be popped from the stack, and $\gamma 2$ will be pushed onto it. M cannot peek at the top of the stack without popping

- If $c = \varepsilon$, the transition can be taken without consuming any input.
- If $\gamma 1 = \varepsilon$, the transition can be taken without checking the stack or popping anything. Note: it's not saying "the stack is empty".
- If $\gamma 2 = \varepsilon$, nothing is pushed onto the stack when the transition is taken.

Example1: A PDA for Balanced Parentheses. Bal={w ∊ {),(}*: the parenthesis are balanced}

M = (K, S, G, , s, A),

where:

K = {s} the states

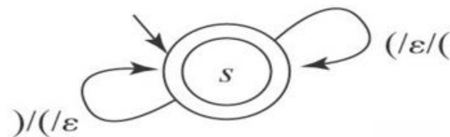
S = {(,)} the input alphabet

Γ = {(} the stack alphabet

A = {s} the accepting state

= { ((s, (,ε), (s, ( )) ----- (1)

((s,), ( ), (s, ε)) ----- (2) }



An Example of Accepting -- Input string = (())()

(S, (())(),ε) |- (S, ())(), ( ) |- (S,))()),(() |- (S,)(), () |- (S, (), ε) |- (S,), () |- (S, ε, ε)

The computation accepts the string ((())() as it runs out of input being in the accepting state S and stack empty.

Transition	State	Unread input	Stack
	S	(())()	ε
1	S	())()	(
1	S	))()	((
2	S	)()	(
2	S	()	ε
1	S	)	(
2	S	ε	ε

Example1 of Rejecting -- Input string = (()))

(S, ())),ε) |- (S, ())),() |- (S,))),(() |- (S,)),() |- (S,), ε)

Transition	State	Unread input	Stack
	S	(()))	ε
1	S	()))	(
1	S	)))	((
2	S	))	(
2	S	)	ε

The computation has reached the final state S and stack is empty, but still the string is rejected because the input is not empty.

Example2 of Rejecting -- Input string = ((())

Transition	State	Unread input	Stack
	S	((())	ε
1	S	(())	(
1	S	())	((
1	S	))	(((
2	S	)	((
2	S	ε	(

(S, ((()),ε) |- (S, (()),(|- (S,( )),(() |- (S,)),((() |- (S,),(() |- (S, ε ,()

The computation has reached the final state S and runs out of input, but still the string is rejected because the stack is not empty.

Example 2: A PDA for $A^nB^n = \{a^nb^n: n \geq 0\}$

M = (K, S, G, , s, A),

where:

K = {s, f} the states

S = {a, b} the input alphabet

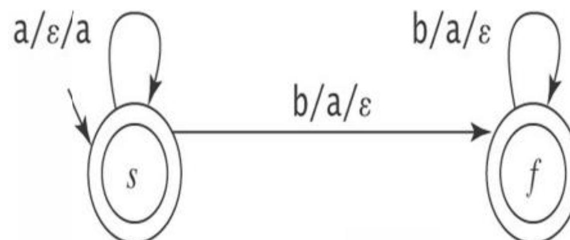
Γ = {a} the stack alphabet

A = {s, f} the accepting state

= { ((s, a, ε), (s, a)) -----(1)

((s, b, a), (f, ε)) -----(2)

((f, b, a), (f, ε)) } -----(3)



An Example of Accepting -- Input string = aabb

(f, aabb,ε) |- (f, abb, a) |- (f, bb, aa) |- (f, b, a) |- (f, ε, ε)

The computation has reached the final state f, the input string is consumed and the stack is empty. Hence the string aabb is accepted.

Example3: A PDA for $\{wcw^R: w \in \{a, b\}^*\}$

M = (K, S, G, , s, A),

where:

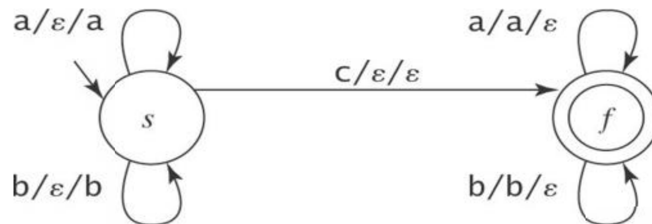
K = {s, f} the states

S = {a, b, c} the input alphabet

Γ = {a, b} the stack alphabet

A = {f} the accepting state

= {((s, a, ε), (s, a) -----(1)
((s, b, ε), (s, b)) -----(2)
((s, c, ε), (f, ε)) -----(3)
((f, a, a), (f, ε)) -----(4)
((f, b, b), (f, ε))} -----(5)



An Example of Accepting -- Input string = abcba
(s, abcba,ε) |- (s, bcba, a) |- (s, cba,ba) |- (f, ba, ba) |- (f, a, a) |- (f, ε, ε)
The computation has reached the final state f, the input string is consumed and the stack is empty. Hence the string abcba is accepted.

Example 4: A PDA for $A^nB^{2n} = \{a^nb^{2n}: n \geq 0\}$
M = (K, S, G, , s, A),
where:

K = {s, f} the states
S = {a, b} the input alphabet
Γ = {a} the stack alphabet
A = {s, f} the accepting state

= { ((s, a, ε), (s, aa)) -----(1)
((s, b, a), (f, ε)) -----(2)
((f, b, a), (f, ε)) } -----(3)

An Example of Accepting -- Input string = aabbbb
(s, aabbbb,ε) |- (s, abbbb, aa) |- (s, bbbb,aaaa) |- (f, bbb, aaa) |- (f, bb, aa) |-(f, b, a) |- (f, ε, ε)

10. Deterministic and Nondeterministic PDAs

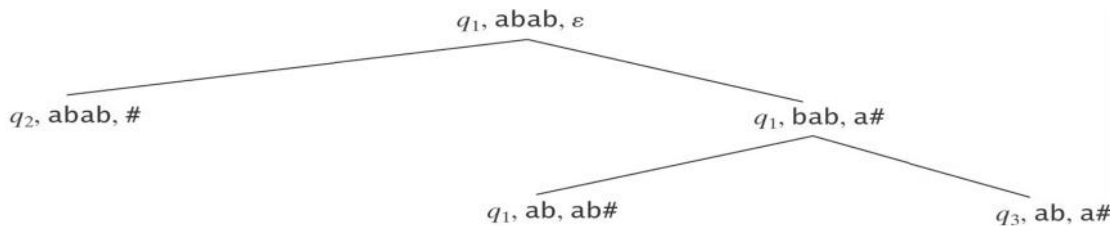
A PDA M is deterministic iff:

- Δ_M contains no pairs of transitions that compete with each other, and
- whenever M is in an accepting configuration it has no available moves.
- If q is an accepting state of M, then there is no transition ((q, e , e) ,(p, a)) for any p or a.

Unfortunately, unlike FSMs, there exist NDPDA s for which no equivalent DPDA exists.

Exploiting Nondeterministic

Previous examples are DPDA, where each machine followed only a single computational path. But many useful PDAs are not deterministic, where from a single configuration there exist multiple competing moves. As in FSMs, easiest way to envision the operation of a NDPDA M is as a tree.



Each node in the tree corresponds to a configuration of M and each path from the root to a leaf node may correspond to one computation that M might perform. The state, the stack and the remaining input can be different along different paths. As a result, it will not be possible to simulate all paths in parallel, the way we did for NDFSMs.

Example 1: PDA for PalEven ={ww^R: w ∈ {a, b}*}.

The L of even length palindrome of a's and b's. = {ε, aa, bb, aaaa, abba, baab, bbbb, ..…….}

M = (K, S, G, Δ, s, A),

where:

K = {s, f} the states

S = {a, b} the input alphabet

Γ = {a, b} the stack alphabet

A = {f} the accepting state

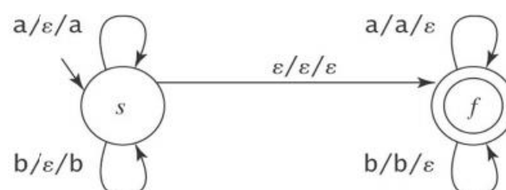
Δ = {((s, a, ε), (s, a)) -----(1)

((s, b, ε), (s, b)) -----(2)

((s, ε, ε), (f, ε)) -----(3)

((f, a, a), (f, ε)) -----(4)

((f, b, b), (f, ε))} -----(5)



Example 2: PDA for $\{w \in \{a, b\}^* : \#_a(w) = \#_b(w)\}$= Equal Numbers of a's and b's.

L = {ε, ab, ba, abba, aabb, baba, bbaa, …….}

M = (K, S, G,   , s, A),

where:

K = {s} the states

S = {a, b} the input alphabet

Γ = {a, b} the stack alphabet

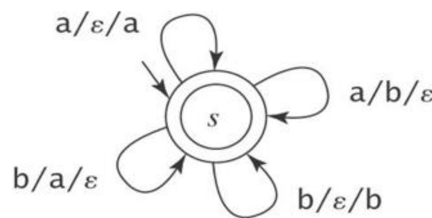
A = {s} the accepting state

={((s, a, ε), (s, a)) -----(1)

((s, b, ε), (s, b)) -----(2)

((s, a, b), (s, ε)) -----(3)

((s, b, a), (s, ε))} -----(4)

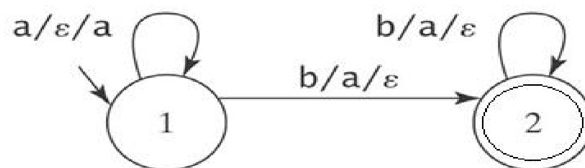


Example 3: The a Region and the b Region are Different. L = $\{a^m b^n : m \quad n; m, n > 0\}$

It is hard to build a machine that looks for something negative, like . But we can break L into two sublanguages: $\{a^m b^n : 0 < n < m\}$ and $\{a^m b^n : 0 < m < n\}$

- If stack and input are empty, halt and reject
- If input is empty but stack is not (m > n) (accept)
- If stack is empty but input is not (m < n) (accept)

Start with the case where n = m

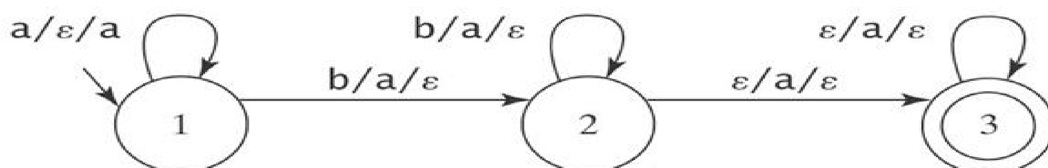


= { ((1, a, ε), (1, a)) -----(1)

((1, b, a), (2, ε)) -----(2)

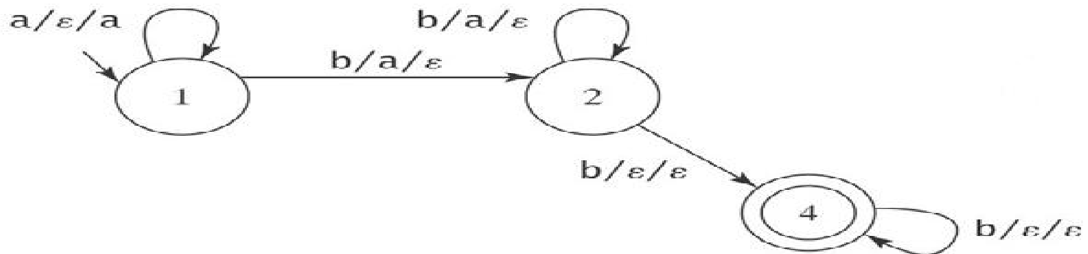
((2, b, a), (2, ε)) } -----(3)

If input is empty but stack is not (m > n) (accept):



= { ((1, a, ε), (1, a))   -----(1)
((1, b, a), (2, ε))   -----(2)
((2, b, a), (2, ε))   -----(3)
((2, ε, a), (3, ε))   -----(4)
((3, ε, a), (3, ε)) } -----(5)

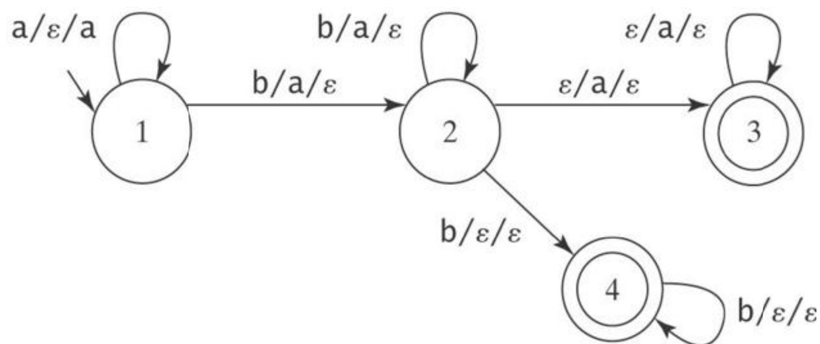
If stack is empty but input is not (m < n) (accept):



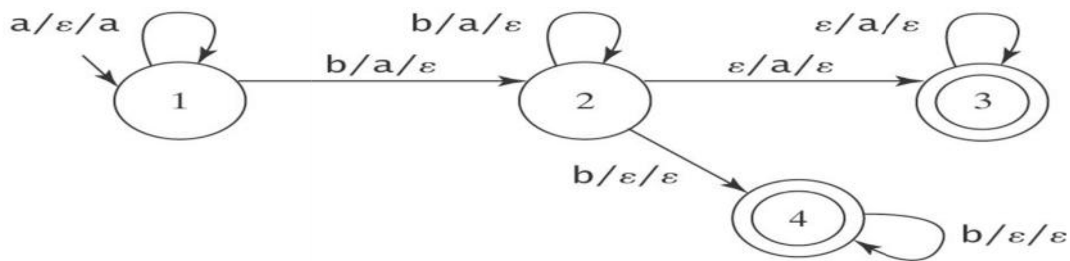
= { ((1, a, ε), (1, a))   -----(1)
((1, b, a), (2, ε))   -----(2)
((2, b, a), (2, ε))   -----(3)
((2, b, ε), (4, ε))   -----(4)
((4, b, ε), (4, ε)) } -----(5)

Putting all together the PDA obtained is
M = (K, S, G,   , s, A),
where:

K = {1,2,3,4}   the states
S = {a, b}   the input alphabet
Γ = {a}   the stack alphabet
A = {3,4}   the accepting state



= { ((1, a, ε), (1, a))   -----(1)
((1, b, a), (2, ε))   -----(2)
((2, b, a), (2, ε))   -----(3)
((2, ε, a), (3, ε))   -----(4)
((3, ε, a), (3, ε)) }   -----(5)
((2, b, ε), (4, ε))   -----(6)
((4, b, ε), (4, ε)) }   -----(7)



Two problems with this M:

1. We have no way to specify that a move can be taken only if the stack is empty.
2. We have no way to specify that the input stream is empty.
3. As a result, in most of its moves in state 2, M will have a choice of three paths to take.

Techniques for Reducing Nondeterminism

We saw nondeterminism arising from two very specific circumstances:

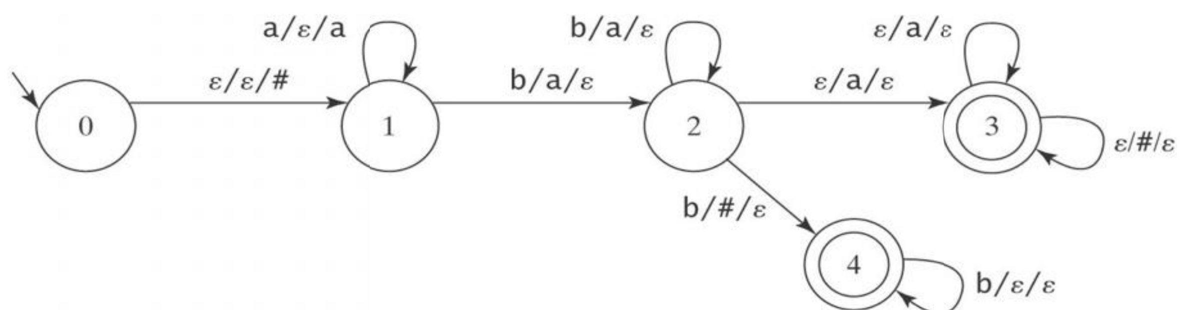
- A transition that should be taken only if the stack is empty competes against one or more moves that require a match of some string on the stack.
- A transition that should be taken only if the input stream is empty competes against one or more moves that require a match against a specific input character.

Case1: A transition that should be taken only if the stack is empty competes against one or more moves that require a match of some string on the stack.

Problem: Nondeterminism could be eliminated if it were possible to check for an empty stack.

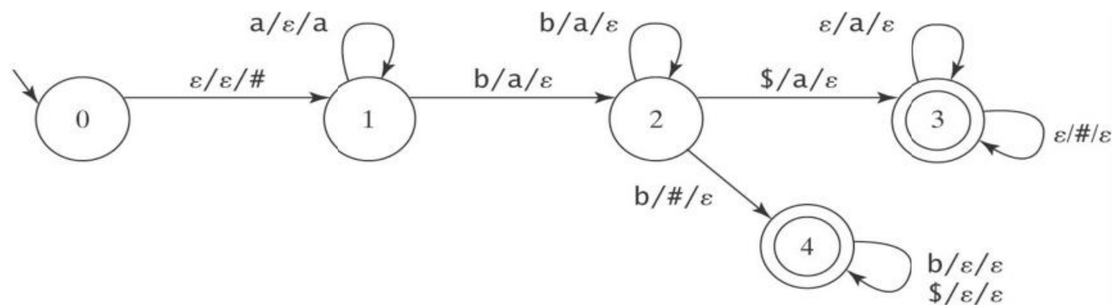
Solution: Using a special bottom-of-stack marker (#)

Before doing anything, push a special character onto the stack. The stack is then logically empty iff that special character (#) is at the top of the stack. Before M accepts a string, its stack must be completely empty, so the special character must be popped whenever M reaches an accepting state.



Now the transition back to state 2 no longer competes with the transition to state 4, which can only be taken when the # is the only symbol on the stack. The machine is still nondeterministic because the transition back to state 2 competes with the transition to state 3.

Case2: A transition that should be taken only if the input stream is empty competes against one or more moves that require a match against a specific input character.
Problem: Nondeterminism could be eliminated if it were possible to check for an empty input stream.
Solution: using a special end-of-string marker ($)
Adding an end-of-string marker to the language to be accepted is a powerful tool for reducing nondeterminism. Instead of building a machine to accept a language L, build one to accept L$, where $ is a special end-of-string marker.



Now the transition back to state 2 no longer competes with the transition to state 3, since the can be taken when the $ is read. The $ must be read on all the paths, not just the one where we need it.

11. Nondeterminism and Halting

Recall Computation C of a PDA M = (K, S, G, , s, A) on a string w is an accepting computation iif C= (s, w, ε) $|-_M*$ (q, ε, ε), for some q ∈ A.
A computation C of M halts iff at least one of the following condition holds:

- C is an accepting computation, or
- C ends in a configuration from which there is no transition in that can be taken.

M halts on w iff every computation of M on w halts. If M halts on w and does not accept, then we say that M rejects w.
For every CFL L, we've proven that there exists a PDA M such that L(M) = L.
Suppose that we would like to be able to:

1. Examine a string and decide whether or not it is L.
2. Examine a string that is in L and create a parse tree for it.
3. Examine a string that is in L and create a parse tree for it in time that is linear in the length of the string.
4. Examine a string and decide whether or not it is in the complement of L.

For every regular language L, there exists a minimal deterministic FSM that accepts it. That minimal DFSM halts on all inputs, accepts all strings that are in L, and rejects all strings that are not in L.

But the facts about CFGs and PDAs are different from the facts about RLs and FSMs.

1. There are context-free languages for which no deterministic PDA exists.
2. It is possible that a PDA may

    not halt,

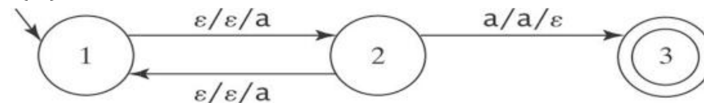
    not ever finish reading its input.

However, for an arbitrary PDA M, there exists M′ that halts and L(M′) = L(M).

There exists no algorithm to minimize a PDA. It is undecidable whether a PDA is minimal.

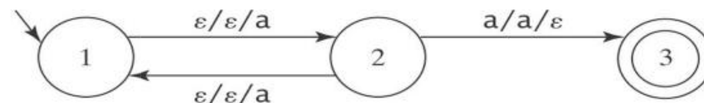
Problem 2 : Let M be a PDA that accepts some language L. Then, on input w, if w ∈ L then M will halt and accept. But if w L then, while M will not accept w, it is possible that it will not reject it either.

Example1: Let S = {a} and consider M =



For L(M) = {a}. The computation (1, a, e) |- (2, a, a) |- (3, e, e) causes M to accept a.

Example2: Consider M =



For L(M) = {aa} or on any other input except a:

(1, aa, e) |- (2, aa, a) |-(1, aa, aa) |- (2, aa, aaa) |- (1, aa, aaaa) |- (2, aa, aaaaa) |- ……..

M will never halt because of one path never ends and none of the paths accepts.

The same problem with NDFSMs had a choice of two solutions.

- Converting NDFSM to and equivalent DFSM using ndfsmtodfsm algorithm.
- Simulating NDFSM using ndfsmsimulate.

Neither of these approaches work on PDAs. There may not even be an equivalent deterministic PDA.

Solution fall into two classes:

- Formal ones that do not restrict the class of the language that are being considered- converting grammar into normal forms like Chomsky or Greibach normal form.
- Practical ones that work only on a subclass of the CFLs- use grammars in natural forms.

12. Alternative Equivalent Definitions of a PDA

PDA M = (K, S, G, , s, A):

1. Allow M to pop and to push any string in G*.
2. M may pop only a single symbol but it may push any number of them.
3. M may pop and push only a single symbol.

M accepts its input w only if , when it finishes reading w, it is in an accepting state and its stack is empty.

There are two alternatives to this:

1. PDA by Final state: Accept if, when the input has been consumed, M lands in an accepting state, regardless of the contents of the stack.
2. PDA by Empty stack: Accept if, when the input has been consumed, the stack is empty, regardless of the state M is in.

All of these definitions are equivalent in the sense that, if some language L is accepted by a PDA using one definition, it can be accepted by some PDA using each of the other definition. For example:- If some language L is accepted by a PDA by Final state then it can be accepted by PDA by Final state and empty stack. If some language L is accepted by a PDA by Final state and empty stack then can be accepted by PDA by Final state.
We can prove by showing algorithms that transform a PDA of one sort into and equivalent PDA of the other sort.
Equivalence

1. Given a PDA M that accepts by accepting state and empty stack, construct a new PDA M′ that accepts by accepting state alone, where L(M′) = L(M).
2. Given a PDA M that accepts by accepting state alone, construct a new PDA M′ that accepts by accepting state and empty stack, where L(M′) = L(M).

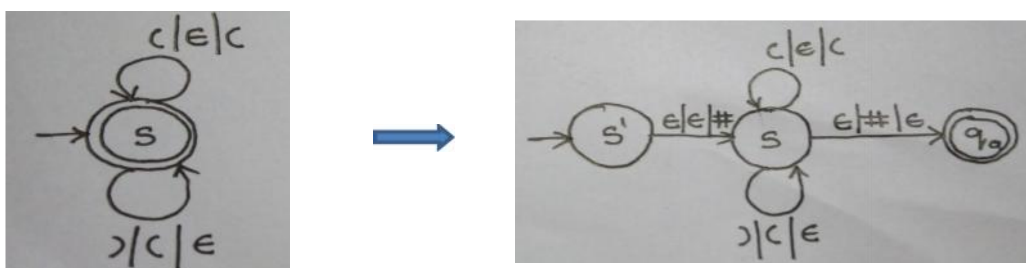
Hence we can prove that M′ and M accept the same strings.

1. Accepting by Final state Alone

Define a PDA M = (K, S, G, , s, A). Accepts when the input has been consumed, M lands in an accepting state, regardless of the contents of the stack. M accepts if C= (s, w, ε) $|\text{-}_M*$ (q, ε, g), for some q $\in$ A.
M′ will have a single accepting state q_a. The only way for M′ to get to q_a will be to land in an accepting state of M when the stack is logically empty. Since there is no way to check that the stack is empty, M′ will begin by pushing a bottom-of-stack marker #, onto the stack. Whenever # is the top symbol of the stack, then stack is logically empty.
The construction proceeds as follows:

1. Initially, let M′ = M.
2. Create a new start state s′.
   Add the transition ((s′, ε , ε),(s, #)),
3. For each accepting state a in M do:
   Add the transition ((a, ε ,#),(q_a, ε)),
4. Make q_a the only accepting state in M′

Example:



It is easy to see that M′ lands in its accepting state(q_a) iff M lands in some accepting state with an empty stack. Thus M′ and M accept the same strings.

2. Accepting by Final state and Empty stack

The construction proceeds as follows:

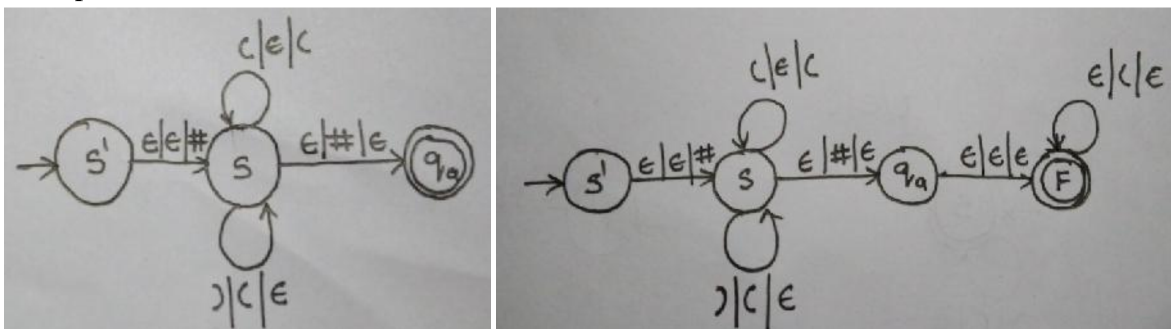
1. Initially, let M′ = M.
2. Create a new accepting state F
3. From each accepting state a in M do:

   Add the transition ((a, ε , ε),(F, ε)),
4. Make F the only accepting state in M′
5. For every element g of ,

   Add the transition to M′ ((F, ε, g), (F, ε)).

In other words, iff M accepts, go to the only accepting state of M and clear the stack. Thus M will accept by accepting state and empty stack iff M accepts by accepting state.

Example:-



Thus M′ and M accept the same strings.

13. Alternatives that are not equivalent to the PDA

We defined a PDA to be a finite state machine to which we add a single stack.

Two variants of that definition, each of which turns out to define a more powerful class of a machine.

1. First variant: add a first-in, first-out (FIFO) queue in place of a stack. Such machines are called tag systems or Post machines.
2. Second variant: add two stacks instead of one. The resulting machines are equivalent in computational power to Turing Machines.

Sl.No	Sample Questions
1.	Define context free grammars and languages.
2.	Show a context-free grammar for each of the following languages L: a) BalDelim = {w : where w is a string of delimeters: (,), [,], {, }, that are properly balanced}. b) $\{a^i b^j : 2i = 3j + 1\}$. c) $\{a^i b^j : 2i \quad 3j + 1\}$. d) $\{a^i b^j c^k : i, j, k \quad 0$ and $(i \quad j$ or $j \quad k)\}$.
3.	Define CFG. Design CFG for the language L={ $a^n b^m$: n m}
4.	Apply the simplification algorithm to simplify the given grammar S AB|AC A aAb | B bA C bCa D AB
5.	Prove the correctness of the grammar for the language: L={w ∈ {a, b}* : $\#_a(w) = \#_b(w)$}.
6.	Define leftmost derivation and rightmost derivation. Given the following CFG. E→ E + T|T T→ T*F|F F→(E)|a|b|c Draw parse tree for the following sentences and also derive the leftmost and rightmost derivations i) (a+b)*c ii) (a) + b*c
7.	Consider the following grammar G: S 0S1 | SS | 10 Show a parse tree produced by G for each of the following strings: a) 010110 b) 00101101
8.	Define ambiguous and explain inherently ambiguous grammars.
9.	Prove whether the given grammar is ambiguous grammar or not. E→ E + E E→ E*E|a|b|c
10.	Prove that the following CFG is ambiguous S→iCtS|iCtSeS|x C→y for the sting iytiytxex
11.	Define Chomsky normal form. Apply the normalization algorithm to convert the grammar to Chomsky normal form. a. S aSa S B B bbC B bb C C cC b. S ABC A aC | D B bB | | A C Ac | | Cc D aa
12.	Define Push down automata (NPDA). Design a NPDA for the CFG given in Question (2).
13.	Design a PDA for the given language.L$, where L = {w ∈ {a, b}* : $\#_a(w) = \#_b(w)$}.
14.	Design a PDA for the language: L={ $a^i b^j c^k$: i+j=k ,i>=0,j>=0}
15.	Design a PDA for the language L={ $a^n b^{2n}$: n>=1}
16.	Design a PDA for the language: L={ $a^i b^j c^k$: i+k=j ,i>=0,k>=0}
17.	Design a PDA for the language: L={ $a^i b^j c^k$: k+j=i ,k>=0,j>=0}